ANL/ALCF-16/3

Camellia v1.0 Manual

Part I

Argonne Leadership Computing Facility

About Argonne National Laboratory
Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

ANL/ALCF-16/3

Camellia v1.0 Manual

Part I

prepared by
Nathan V. Roberts
Argonne Leadership Computing Facility, Argonne National Laboratory

September 28, 2016

Camellia v1.0 Manual
Part I

Nathan V. Roberts

September 28, 2016

Contents

Acknowledgments

This work was supported by the Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

I gratefully acknowledge the support of Argonne and the Argonne Leadership Computing Facility during the postdoctoral work that allowed much of the development documented in this manual. I thank Brendan Keith for critical feedback regarding the manuscript.

I thank Paul Fischer for advice regarding geometric multigrid preconditioning, and Jesse Chan for collaborating on the geometric multigrid work. I thank Jesse Chan, Truman Ellis, and Brendan Keith for their contributions to Camellia development.

I thank Leszek Demkowicz for ongoing collaborations regarding discontinuous Petrov-Galerkin methods, and for his support during my PhD. work and since.

Chapter 1

Introduction

Camellia [14] began as an effort to simplify implementation of efficient solvers for the discontinuous Petrov-Galerkin (DPG) finite element methodology of Demkowicz and Gopalakrishnan [6, 7]. Since then, the feature set has expanded, to allow implementation of traditional continuous Galerkin methods, as well as discontinuous Galerkin (DG) methods, hybridizable DG (HDG) methods [5], first-order-system least squares (FOSLS) [3], and the primal DPG method [8].

This manual serves as an introduction to using Camellia. We begin, in Section 1.1, by describing some of the core features of Camellia. In Section 1.2 we provide an outline of the manual as a whole.

1.1 Core Features

Camellia provides mechanisms for rapid specification for many of the standard parts of finite element methods, including:

- bilinear formulations,
- boundary conditions, and
- material and load data,

as well as the inner product on the test space, a requirement for DPG solves. Camellia supports both field and trace variables—that is, variables defined on the mesh volume as well as those defined on the mesh skeleton. For both of these, vector and scalar variables are supported. Basis functions conforming

to the exact sequence—that is, H^1, $H(\text{div})$, $H(\text{curl})$, and L^2—are provided on each of the supported topologies, which include lines in 1D, triangle and quadrilaterals in 2D, and hexahedra and tetrahedra in 3D. Support for regular h-refinements of all these—with the exception of tetrahedra in 3D—is provided.[1] Essentially arbitrary polynomial orders are supported for hypercube topologies (some limitations are imposed on simplices).

Because Camellia began as a research tool investigating DPG in the context of problems with relatively simple geometries, support for curvilinear elements is currently limited to 2D geometries specified parametrically (a somewhat homegrown approach). While some examples can be found in the source code, we do not discuss curvilinear geometry in this manual; we expect to support curvilinear geometry in arbitrary dimensions in a fashion similar to other finite element codes (likely through nodal weights on each element to specify geometry isoparametrically). This should be a relatively simple extension of the current straight-edged implementation, and we hope to have support for this in the near future.

At present, Camellia supports only real-valued, double-precision variables—though it is worth noting that complex problems can be implemented by introducing a real-valued variable for each component of complex variables. We would like to extend Camellia to support templated scalar types in a future release—this will allow direct support of complex variables, as well as single-precision and quad-precision variables.

Essentially all computationally-intensive features support distributed computation using MPI. Load-balancing is implemented using Zoltan. For DPG systems, a robust iterative solver using geometric multigrid is provided. Support for static condensation[2] for ultraweak DPG systems is also provided.

1.2 Structure of this Manual

In Chapter 2, we cover a few preliminary concepts and conventions that will be useful in the code examples. The remainder of the manual is driven by examples. The first part of the manual focuses on DPG solvers, starting

[1] Support for tetrahedral refinements is a relatively simple extension that we hope to add in the near future.

[2] Static condensation eliminates local degrees of freedom, to reduce the size of the global system at the cost of some local computation. In the present release of Camellia, we support element-local elimination of discontinuous field variables.

in Chapter 3, wherein we demonstrate the implementation of an adaptive Stokes solver for the steady 2D lid-driven cavity flow problem, using DPG (a brief introduction to DPG is also provided there). The third chapter extends that implementation to support Navier-Stokes (and discusses one approach to DPG in the context of nonlinear problems). These two chapters include the full implementation of the appropriate variational formulations; in Chapter 5 we discuss usage of some formulations whose implementations are already provided in Camellia for Poisson, Stokes, and Navier-Stokes—these can be a much quicker way to get started.

Chapter 6 discusses supported options for the global linear solve, including KLU, MUMPS, SuperLUDist, and Camellia's built-in support for geometric multigrid.

There are many further topics that we plan to address in detail in future updates to this manual; we publish this as "Part I." In Appendix A, we cover in a summary fashion several of these further topics, including pointers on where to look in the Camellia distribution for example implementations.

Chapter 2

Some Preliminaries

In this chapter, we cover background material of several kinds. We begin by discussing a few coding conventions used in Camellia and useful in Camellia drivers; then we define some terminology, and give a brief introduction to the DPG methodology and related concepts. We conclude the chapter with an introduction to several core classes in Camellia, which will be used throughout the rest of the manual.

2.1 Coding Conventions

In this section, we begin by discussing reference-counted pointers, then turn to a class that assists with processing command-line arguments. We then give a brief introduction to several C++11 features used throughout Camelllia.

2.1.1 Reference-Counted Pointers in Camellia

Camellia makes extensive use of reference-counted pointers (RCPs), a mechanism for simplifying memory management. An RCP "knows" how many references there are to it. Once the number of references falls to zero, the RCP can safely deallocate the memory it points to. Camellia uses the Teuchos `RCP` class to define its reference-counted pointers. To define a Teuchos RCP pointing to a new object of type `MyClass`, one might write

```
Teuchos::RCP<MyClass> myPtr = Teuchos::rcp(new MyClass());
```

Here, `Teuchos::RCP<MyClass>` identifies a type templated on the `MyClass` class. The `Teuchos::rcp()` method creates a new object of this type,

pointing to the new object created by `new MyClass()`. At this point, the reference count is 1, because `myPtr` refers to the object, and nothing else does. Because, especially for longer class names, `Teuchos::RCP<MyClass>` can be a lot to type, Camellia adopts the convention of defining type names (using C++ typedefs) of the form `MyClassPtr` to refer to the type `Teuchos::RCP<MyClass>`.

Often, Camellia classes provide static constructors that return an RCP; for example, the `Solution::solution()` constructors return a `SolutionPtr` object. In these cases, there is no need either for `Teuchos::rcp()` or for the `new` operator.

There are some issues that can arise with RCPs—one particular concern is circular references, which can result in memory leaks. Because these are not likely to come up in standard usage of Camellia, we do not discuss them here; however, we refer the interested reader to the *Teuchos::RCP Beginner's Guide*, available from Sandia [1].

2.1.2 CommandLineProcessor

Teuchos provides a class, `CommandLineProcessor`, which assists in processing command-line arguments. For instance, the following code defines variables `polyOrder` and `useIterativeSolver`, with default values of `5` and `true`:

```
using namespace Teuchos;
CommandLineProcessor cmdp(false,true);
int polyOrder = 5;
bool iterative = true; // otherwise, use direct solver
cmdp.setOption("polyOrder", &polyOrder );
cmdp.setOption("iterative", "direct", &iterative);
auto parseResult = cmdp.parse(argc,argv);
if (parseResult != CommandLineProcessor::PARSE_SUCCESSFUL)
{
  return -1;
}
```

If this code were placed in the `main()` method of a driver `MyDriver`, then the driver could be invoked as follows:

```
./MyDriver --direct --polyOrder=8
```

which would initialize `polyOrder` to 8, and `iterative` to `false`. Moreover, invoking

```
./MyDriver --help
```

will print to console all options available.

2.1.3 Some Useful C++11 Features

Here, we briefly describe a few features provided by C++11, of which we make extensive use in Camellia, and which may prove useful in developing Camellia drivers:

- the `auto` keyword
- initializer lists, and
- range-based for loops.

2.1.4 The `auto` Keyword

In the `CommandLineProcessor` example code above, we had a line:

```
auto parseResult = cmdp.parse(argc,argv);
```

This declares `parseResult` to be of whatever type `cmdp.parse()` returns; the C++ compiler will determine the type from context. This is particularly convenient when the type returned is long and/or complicated, as sometimes happens with templated code.

Initializer Lists

Many Camellia methods take C++ standard template library containers such as `vector` or `set` as arguments. C++11 provides a new interface for constructing such containers, called *initializer lists*, in which values are provided explicitly on construction. For example, one can write the following:

```
vector<double> dims = {1.0, 3.5, 8.0};
vector<vector<double>> points = {{0,0,0},{0,1,0},{0,1,1}};
```

Range-Based For Loops

C++11 also provides a new interface for convenient iteration over the members of a container, in which one may write code of the form `for (element : container)` to iterate over the elements in the container. Thus, if we have the `points` container defined as above, we might write:

```
for (auto point : points)
{
  cout << "x coord is: " << point[0] << endl;
}
```

to print out the x coordinate of each point.

2.2 Terminology

Here, we define a few terms that we employ throughout this manual. DPG solutions can involve not only variables defined throughout the domain but also variables that are defined only on the mesh skeleton. We call solution variables defined throughout the domain *field* variables; often, these will be discontinuous across inter-element interfaces. We call variables that are defined only on the mesh skeleton *trace* variables; this is because mathematically these arise by taking the trace of some combination of field variables. When the term traced involves an outward-facing normal on each element, we call this a *flux* variable. Our convention is to place hats on variables to show that they are trace variables—when there is a corresponding field variable, we use the same symbol: $\widehat{u}$ is a trace corresponding to field u.

For simplicity, we classify DPG formulations into two types: *ultraweak* and *primal* formulations. In ultraweak formulations, all derivatives are moved from the trial space onto the test space through elementwise integration by parts, and each field variable is discontinuous (mathematically, only L^2 regularity is assumed in the field variables). All other DPG formulations are known as primal formulations: these include at least one field variable of higher regularity. Note that primal DPG formulations usually do include at least one flux or trace variable.[1]

[1]It is worth noting that what we are here calling "primal" DPG formulations include what are sometimes called mixed DPG formulations elsewhere. For an example of the range of DPG formulations available for a given problem, see Carstensen et al. [4], especially Section 6, in which several formulations for Maxwell are discussed.

2.3 A Brief Introduction to DPG

In this section, we give a very brief overview of the DPG methodology. DPG stands for *discontinuous Petrov-Galerkin*. Here, *Petrov*-Galerkin means that the test space is allowed to differ from the trial space. In particular, we compute the test space on the fly in such a way that the inf-sup condition is automatically satisfied—this gives us the property that DPG is automatically stable and generates a system matrix that is symmetric (Hermitian) positive definite.

The test space for an element K is constructed as follows. Suppose that we are given a bilinear form $b(\cdot,\cdot) : U_h(K) \times V_h(K) \to \mathbb{R}$ defined on some discrete trial space $U_h(K)$ and discrete test space $V_h(K)$. We require that the discrete test space has dimension at least as large as the trial space (in practice, we may use a test space that is considerably larger). Suppose that we are also given an inner product $(\cdot,\cdot)_V$ on the test space. Now, if we take $e \in U_h(K)$, then $b(e,\cdot) : V \to \mathbb{R}$ is a functional on the test space. By the Riesz Representation Theorem, there exists a unique $v_e \in V_h(K)$ such that $(v_e,\cdot)_V = b(e,\cdot)$. By prescribing a v_e for each $e \in U_h(K)$, we construct a test space that has the same dimension as $U_h(K)$. Moreover, by testing with these, we generate a system matrix that is Hermitian positive definite:

$$b(e_i, v_{e_j}) = (v_{e_i}, v_{e_j})_V = \overline{(v_{e_j}, v_{e_i})_V} = \overline{b(e_j, v_{e_i})}.$$

We emphasize that the above construction is performed elementwise. The test space is thus allowed to be discontinuous across element interfaces; this is the reason the method is called discontinuous. In the case of ultraweak formulations, we additionally allow all field variables to be discontinuous. In contrast to DG methods, in DPG we always enforce *some* continuity requirements—for ultraweak formulations, continuity is enforced on the trace and flux variables.

The Energy Norm and the Test Space Inner Product

As is obvious from the above, the behavior of DPG depends closely on the choice of inner product on the test space. More precisely, if we define the *energy norm* $\|\cdot\|_E$ on the trial space by

$$\|u\|_E = \sup_{v \in V, v \neq 0} \frac{b(u,v)}{(v,v)_V^{1/2}},$$

then for an exact solution u, the DPG solution u_h minimizes the error $\|u - u_h\|_E$. Thus, if we select a different inner product on V, we will change the norm in which we minimize the error. A natural question arises: can we define the V inner product in such a way that $\|\cdot\|_E$ is equivalent to some norm of interest on U, with modest equivalence constants? Generally, determining such an inner product will require problem-specific analysis, but if the norm of interest is the L^2 norm of the field variables, and we are using an ultraweak formulation, then using the so-called *adjoint graph norm* on the test space is appropriate. A full treatment of the graph norm can be found in Roberts et al. [15]; here, we simply describe the algorithm which may be used to generate the graph norm.

We begin by rewriting the bilinear form: for each field variable u_i, we collect all the test variables that are integrated against it. We can then write the u_i portion of the bilinear form as $(u_i, A_i v)$, where $(\cdot, \cdot)$ is the usual L^2 inner product, v is a group variable, and A_i is some linear differential operator on V. The graph norm is then defined by:

$$\|v\|^2_{\text{graph}} = \sum_i \|A_i v\|^2_{L^2} + \|v\|^2_{L^2}\,.$$

The inner product on the test space can then be taken to be the one that generates this norm. (The $\|v\|^2_{L^2}$ term effectively "controls" the terms involving traces and fluxes; one may introduce a small weight on this term to improve the constants that govern the desired equivalence between the L^2 norm and the energy norm.)

2.3.1 An Example: Ultraweak Poisson

To take one example, consider the Poisson problem:

$$\Delta u = f$$

on a domain Ω with some mesh Ω_h. To get the ultraweak DPG formulation, we first introduce a new unknown $\boldsymbol{\sigma} = \nabla u$ and rewrite as a system of first-order equations:

$$\begin{aligned} \nabla \cdot \boldsymbol{\sigma} &= f, \\ \boldsymbol{\sigma} - \nabla u &= 0. \end{aligned}$$

We now multiply by test functions v and $\boldsymbol{\tau}$, and integrate by parts on each element K:

$$\begin{aligned}(-\boldsymbol{\sigma}, \nabla v)_K + \langle \boldsymbol{\sigma} \cdot \boldsymbol{n}, v \rangle_{\partial K} &= (f, v)_K, \\ (\boldsymbol{\sigma}, \boldsymbol{\tau})_K + (u, \nabla \cdot \boldsymbol{\tau}) - \langle u, \boldsymbol{\tau} \cdot \boldsymbol{n} \rangle_{\partial K} &= 0,\end{aligned}$$

where $(\cdot, \cdot)_K$ is the L^2 inner product on element K. We now introduce a new trace unknown $\widehat{u}$ and a new flux unknown $\widehat{\sigma}_n$ and sum both equations to get our bilinear form b and load l:

$$b(\cdot, \cdot) = (-\boldsymbol{\sigma}, \nabla v)_K + \langle \widehat{\sigma}_n, v \rangle_{\partial K} \tag{2.3.1}$$

$$+ (\boldsymbol{\sigma}, \boldsymbol{\tau})_K + (u, \nabla \cdot \boldsymbol{\tau})_K - \langle \widehat{u}, \boldsymbol{\tau} \cdot \boldsymbol{n} \rangle_{\partial K} = (f, v)_K = l(\cdot). \tag{2.3.2}$$

Now, we may rewrite b as follows:

$$b(\cdot, \cdot) = (\boldsymbol{\sigma}, \boldsymbol{\tau} - \nabla v)_K + (u, \nabla \cdot \boldsymbol{\tau})_K + \{\text{boundary terms}\},$$

which leads us to the following graph norm on the test space:

$$\|(v, \boldsymbol{\tau})\|_V^2 = \|\boldsymbol{\tau} - \nabla v\|^2 + \|\nabla \cdot \boldsymbol{\tau}\|^2 + \|v\|^2 + \|\boldsymbol{\tau}\|^2 . \tag{2.3.3}$$

(Strictly speaking, the $\|\boldsymbol{\tau}\|^2$ term is not required, as the required equivalence for the corresponding trace term—$\langle \widehat{u}, \boldsymbol{\tau} \cdot \boldsymbol{n} \rangle_{\partial K}$—will be achieved by the $\|\boldsymbol{\tau} - \nabla v\|^2$ term alone; we include the $\|\boldsymbol{\tau}\|^2$ term here to be consistent with the algorithm described above.)

2.4 Core Classes in Camellia

There are four types of variables defined in Camellia: test, field, trace, and flux variables. Fields, traces, and fluxes are always trial (solution) variables. Variables are implemented by the `Var` class; the `VarFactory` class allows generation and tracking of variables. To create a `VarFactory` object that tracks the variables from the Poisson ultraweak formulation in (2.3.1), we can do the following:

```
VarFactoryPtr vf = VarFactory::varFactory();
// field variables:
VarPtr u = vf->fieldVar("u", L2);
VarPtr sigma = vf->fieldVar("sigma", VECTOR_L2);
// trace variable:
```

```
VarPtr u_hat = vf->traceVar("u_hat", HGRAD);
// flux variable:
VarPtr sigma_n = vf->fluxVar("sigma_n", L2);
// test variables:
VarPtr v = vf->testVar("v", HGRAD);
VarPtr tau = vf->testVar("tau", HDIV);
```

Functions in Camellia are implemented by the `Function` class. Functions can depend not only on spatial coordinates, but also on the mesh. There are built-in operator overloads for basic arithmetic operations on `FunctionPtr` instances, so that if one has `FunctionPtr`s `f1`, `f2`, `f3`, one can write

```
FunctionPtr f4 = 3 + (f1 + 5 * f2) / f3;
```

with the expected result in `f4`. Functions generally compute values on many points at once, amortizing the overhead associated with method calls. Several static constructors are provided in the `Function` class to make working with functions simpler. Among these are the following:

- `constant()`—returns a scalar or vector function with constant value.
- `h()`—returns a mesh-dependent function corresponding to the diameter of the current element.
- `normal()`—returns a mesh-dependent function corresponding to the outward-facing normal of the current element
- `sqrt(f)`—returns a function with value corresponding to the square root of `f`.
- `solution(var,soln)`—returns a mesh-dependent function with values corresponding to the `var` component of solution `soln`.
- `xn(n)`—returns the function x^n, where n is an integer.
- `yn(n)`—returns the function y^n, where n is an integer.
- `zn(n)`—returns the function z^n, where n is an integer.
- `vectorize(f1,f2[,f3])`—returns a vector-valued function with the provided functions as components.

The `Function` class also provides virtual methods corresponding to differential operations; thus, one can take the x-derivative of some `Function` object; the result is also a `Function`. For some functions, derivatives would be undefined; in these cases, differential operations will return a null function.

A key concept in Camellia is that of the *linear term*, implemented by the `LinearTerm` class. This is an expression that is linear in the trial or test variables. A linear term can be integrated on a specified discretization of a domain, producing a vector (one entry per degree of freedom). `LinearTerm` instances can also be integrated against each other on an element or over a mesh, producing a matrix (one row and column for each degree of freedom). There are also mechanisms for substituting function values for variables, producing a function from a linear term.

`LinearTerm` objects consist of a `Function` weight and a `Var` instance. There are overloads provided so that given a `FunctionPtr f` and a `VarPtr v`, one may write

```
LinearTermPtr lt = f * v;
```

Linear terms are used in several places in Camellia. Bilinear forms are built up out of (trial, test) pairings of linear terms; inner products are built out of terms on either the trial or the test space. These are implemented by the `BF` and `IP` classes, respectively. To make this clearer, suppose that we have defined the `VarFactory vf` and `VarPtr`s `u`, `sigma`, `u_hat`, `sigma_n`, `v`, and `tau` for the ultraweak Poisson formulation as above. We may then create a `BF` object with these variables as follows:

```
BFPtr bf = BF::bf(vf);
```

To define the bilinear form as in (2.3.1), we can do the following:

```
bf->addTerm(-sigma, v->grad());
bf->addTerm(sigma_n, v);

FunctionPtr n = Function::normal();
bf->addTerm(sigma, tau);
bf->addTerm(u, tau->div());
bf->addTerm(-u_hat, tau * n);
```

Furthermore, to define a test space inner product corresponding to the graph norm (2.3.3) listed above, we may write the following:

```
IPPtr ip = IP::ip();
ip->addTerm(tau - v->grad());
```

```
ip->addTerm(tau->div());
ip->addTerm(v);
ip->addTerm(tau);
```

Note that in contrast to the bilinear form, here we assume symmetry of the form—only one linear term is provided to `IP::addTerm()`; each `addTerm()` invocation corresponds to one of the squared terms in (2.3.3).

Chapter 3

Stokes Cavity Flow Using DPG

This chapter demonstrates the implementation of a DPG variational formulation for the Stokes equations, and how to use this to write an adaptive solver for the lid-driven cavity flow problem. We begin in Section 3.1 by showing the mathematical derivation of the so-called ultraweak variational formulation for the velocity-gradient-pressure Stokes system. In Section 3.2, we show how to implement this formulation in Camellia. We then define the cavity flow problem in Section 3.3; we show how to implement an adaptive solver for this problem in Section 3.4.

3.1 The DPG Ultraweak Formulation

We want to solve the Stokes equations in domain $\Omega \subset \mathbb{R}^2$:

$$\begin{aligned} -\mu\Delta\boldsymbol{u} + \nabla p &= \boldsymbol{f} && \text{in } \Omega, \\ \nabla \cdot \boldsymbol{u} &= 0 && \text{in } \Omega, \\ \boldsymbol{u} &= \boldsymbol{u}_D && \text{on } \partial\Omega, \end{aligned}$$

where μ is (constant) viscosity, $\boldsymbol{f}$ is a vector forcing function, and the unknowns are pressure p and velocity $\boldsymbol{u}$ velocity, with some prescribed velocity data $\boldsymbol{u}_D$ on the domain boundary.

To derive an ultraweak formulation, we take the following steps:

1. Introduce variables and equations to produce a first-order system.
2. Multiply by test functions and integrate.

3. Integrate by parts *elementwise* to move all derivatives to the test space.

4. Introduce new *trace* unknowns on the mesh skeleton corresponding to the trial space boundary terms arising from integration by parts.

We take each of these steps for the Stokes system in turn. First, we introduce a new unknown $\boldsymbol{\sigma} = \mu \nabla \boldsymbol{u}$, giving us the system

$$\begin{aligned} -\nabla \cdot \boldsymbol{\sigma} + \nabla p &= \boldsymbol{f} && \text{in } \Omega, \\ \frac{1}{\mu}\boldsymbol{\sigma} - \nabla \boldsymbol{u} &= 0 && \text{in } \Omega, \\ \nabla \cdot \boldsymbol{u} &= 0 && \text{in } \Omega, \\ \boldsymbol{u} &= \boldsymbol{u}_D && \text{on } \partial\Omega. \end{aligned}$$

Because the new unknown is a gradient of the velocity, this is known as the velocity-gradient-pressure (VGP) system. Other choices for first-order systems include velocity-vorticity-pressure (VVP) and velocity-stress-pressure (VSP).[1] We choose VGP principally because we have performed careful analysis of the corresponding DPG formulation [15].

Now, assume that we have partitioned the domain Ω into a mesh Ω_h. We now multiply by test functions—vector $\boldsymbol{v}$, tensor $\boldsymbol{\tau}$, and scalar q—and integrate each equation by parts on each element.

$$\begin{aligned} (\boldsymbol{\sigma} - p\boldsymbol{I}, \nabla \boldsymbol{v})_{\Omega_h} - \langle (\boldsymbol{\sigma} - p\boldsymbol{I})\boldsymbol{n}, \boldsymbol{v} \rangle_{\partial\Omega_h} &= (\boldsymbol{f}, \boldsymbol{v})_{\Omega_h}, \\ (\boldsymbol{u}, \nabla q)_{\Omega_h} - \langle \boldsymbol{u} \cdot \boldsymbol{n}, q \rangle_{\partial\Omega_h} &= 0, \\ \left(\frac{1}{\mu}\boldsymbol{\sigma}, \boldsymbol{\tau}\right)_{\Omega_h} + (\boldsymbol{u}, \nabla \cdot \boldsymbol{\tau})_{\Omega_h} - \langle \boldsymbol{u}, \boldsymbol{\tau}\boldsymbol{n} \rangle_{\partial\Omega_h} &= 0, \\ \boldsymbol{u} &= \boldsymbol{u}_D \text{ on } \partial\Omega. \end{aligned}$$

The ultraweak variational formulation can be understood as one in which all the variables defined on the volume (the *field* variables)—here, $\boldsymbol{u}, p$, and $\boldsymbol{\sigma}$—are allowed to be discontinuous across element interfaces. Because of this—and for related concerns about the functional setting of the formulation—we need to do something to treat the terms in our system that are evaluated on $\partial\Omega_h$. We therefore introduce new *trace* unknowns: a

[1] DPG formulations for these Stokes systems are derived in Nathan V. Roberts's dissertation [13].

velocity trace $\widehat{\boldsymbol{u}}$, corresponding to $\boldsymbol{u}$, and $\widehat{\boldsymbol{t}}_n$, corresponding to $(\boldsymbol{\sigma} - p\boldsymbol{I})\boldsymbol{n}$,[2] on $\partial\Omega_h$. We also rewrite the boundary conditions in terms of the trace unknowns. We then have:

$$\begin{aligned}
(\boldsymbol{\sigma} - p\boldsymbol{I}, \nabla\boldsymbol{v})_{\Omega_h} - \langle \widehat{\boldsymbol{t}}_n, \boldsymbol{v}\rangle_{\partial\Omega_h} &= (\boldsymbol{f}, \boldsymbol{v})_{\Omega_h}, \\
(\boldsymbol{u}, \nabla q)_{\Omega_h} - \langle \widehat{\boldsymbol{u}} \cdot \boldsymbol{n}, q\rangle_{\partial\Omega_h} &= 0, \\
\left(\frac{1}{\mu}\boldsymbol{\sigma}, \boldsymbol{\tau}\right)_{\Omega_h} + (\boldsymbol{u}, \nabla \cdot \boldsymbol{\tau})_{\Omega_h} - \langle \widehat{\boldsymbol{u}}, \boldsymbol{\tau}\boldsymbol{n}\rangle_{\partial\Omega_h} &= 0, \\
\widehat{\boldsymbol{u}} &= \boldsymbol{u}_D \text{ on } \partial\Omega.
\end{aligned}$$

Summing the equations, we have our Stokes bilinear formulation:

$$\begin{aligned}
b_{\text{Stokes}}(u, v) \stackrel{\text{def}}{=} &(\boldsymbol{\sigma} - p\boldsymbol{I}, \nabla\boldsymbol{v})_{\Omega_h} - \langle \widehat{\boldsymbol{t}}_n, \boldsymbol{v}\rangle_{\partial\Omega_h} \\
&+ (\boldsymbol{u}, \nabla q)_{\Omega_h} - \langle \widehat{\boldsymbol{u}} \cdot \boldsymbol{n}, q\rangle_{\partial\Omega_h} \\
&+ \left(\frac{1}{\mu}\boldsymbol{\sigma}, \boldsymbol{\tau}\right)_{\Omega_h} + (\boldsymbol{u}, \nabla \cdot \boldsymbol{\tau})_{\Omega_h} - \langle \widehat{\boldsymbol{u}}, \boldsymbol{\tau}\boldsymbol{n}\rangle_{\partial\Omega_h} = (\boldsymbol{f}, \boldsymbol{v})_{\Omega_h}.
\end{aligned} \tag{3.1.1}$$

3.2 Stokes Bilinear Form Implementation

We are now ready to write code that defines the formulation we derived above. We proceed in several steps:

1. Create a `VarFactory` object; this keeps track of the variables in play.
2. Define field variables.
3. Define trace and flux variables.
4. Define test variables.
5. Create a `BF` (bilinear form) object.
6. Add terms to the `BF` corresponding to the formulation we derived above.

First, create the `VarFactory` object:

```
VarFactoryPtr vf = VarFactory::varFactory();
```

[2] $\widehat{\boldsymbol{t}}_n$ is sometimes called a pseudo-traction; $\boldsymbol{\sigma}$ is sometimes referred to as a pseudo-stress.

Next, define the field variables. We have vector $\boldsymbol{u}$, scalar p, and tensor $\boldsymbol{\sigma}$, each of which is in L^2 on each element (that is, they are allowed to be discontinuous across element interfaces). Camellia provides support for vector L^2 variables, but at present it does not yet support tensor-valued L^2 variables. For simplicity here, we assume two space dimensions and define scalar variable p and vector variables $\boldsymbol{u}, \boldsymbol{\sigma}_1$, and $\boldsymbol{\sigma}_2$.

```
VarPtr u = vf->fieldVar("u", VECTOR_L2);
VarPtr p = vf->fieldVar("p", L2);
VarPtr sigma1 = vf->fieldVar("sigma_1", VECTOR_L2);
VarPtr sigma2 = vf->fieldVar("sigma_2", VECTOR_L2);
```

We have two kinds of trace variables in our formulation: the $\widehat{u}$ variables are traces of H^1, while the $\widehat{t}_n$ variables are normal traces of $H(\mathrm{div})$. Because the direction of the outward normal on one element's face will be the opposite of that on its neighbor, we need to weight the $\widehat{t}_n$ variables with ± 1 during integration. To distinguish this case from the H^1-variable case, we call variables that involve normal traces *fluxes*. We define two H^1 traces, $\widehat{u}_1$ and $\widehat{u}_2$, and two fluxes, $\widehat{t}_{1n}$ and $\widehat{t}_{2n}$:

```
VarPtr u1_hat = vf->traceVar("u1_hat", HGRAD);
VarPtr u2_hat = vf->traceVar("u2_hat", HGRAD);
VarPtr tn1_hat = vf->fluxVar("tn1_hat", L2);
VarPtr tn2_hat = vf->fluxVar("tn2_hat", L2);
```

For trace variables, the function space designations—here, `HGRAD` for $\widehat{u}$ and `L2` for $\widehat{t}_n$—determine the kind of continuity that will be enforced on element sides (edges in 2D and faces in 3D). `HGRAD` means that vertex continuity will be enforced; `L2` means that the solution may be discontinuous at vertices in 2D and at vertices and edges in 3D.

We are now ready to define our test variables. Looking at equation (3.1.1), we need a scalar q of which we take the gradient; therefore, this should be placed in an H^1 space. We also need a vector $\boldsymbol{v}$ of which we take the gradient; we implement this as H^1 scalar variables v_1, v_2. We also need vectors $\boldsymbol{\tau}_1, \boldsymbol{\tau}_2$ complementing $\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2$; we need to take divergences of the $\boldsymbol{\tau}_i$, so these should be placed in an $H(\mathrm{div})$ space. The following lines define the test variables:

```
VarPtr q = vf->testVar("q", HGRAD);
VarPtr v1 = vf->testVar("v_1", HGRAD);
VarPtr v2 = vf->testVar("v_2", HGRAD);
VarPtr tau1 = vf->testVar("tau_1", HDIV);
VarPtr tau2 = vf->testVar("tau_2", HDIV);
```

Next, we create a new `BF` object; this will define our bilinear form in terms of the variables in our `VarFactory`:

```
BFPtr stokesBF = BF::bf(vf);
```

We then add terms corresponding to the terms in equation (3.1.1). To start, we add terms corresponding to $(\boldsymbol{\sigma} - p\boldsymbol{I}, \nabla \boldsymbol{v})_{\Omega_h} - \langle \widehat{\boldsymbol{t}}_n, \boldsymbol{v} \rangle_{\partial\Omega_h}$:

```
stokesBF->addTerm(sigma1, v1->grad());
stokesBF->addTerm(-p, v1->dx());
stokesBF->addTerm(-tn1_hat, v1);

stokesBF->addTerm(sigma2, v2->grad());
stokesBF->addTerm(-p, v2->dy());
stokesBF->addTerm(-tn2_hat, v2);
```

Next, we add terms corresponding to $(\boldsymbol{u}, \nabla q)_{\Omega_h} - \langle \widehat{\boldsymbol{u}} \cdot \boldsymbol{n}, q \rangle_{\partial\Omega_h}$:

```
stokesBF->addTerm(u, q->grad());
// get a normal function:
FunctionPtr n = Function::normal();
stokesBF->addTerm(-u1_hat * n->x() - u2_hat * n->y(), q);
```

Finally, we add terms corresponding to $\left(\frac{1}{\mu}\boldsymbol{\sigma}, \boldsymbol{\tau}\right)_{\Omega_h} + (\boldsymbol{u}, \nabla \cdot \boldsymbol{\tau})_{\Omega_h} - \langle \widehat{\boldsymbol{u}}, \boldsymbol{\tau}\boldsymbol{n} \rangle_{\partial\Omega_h}$:

```
double mu = 1.0; // unit viscosity
stokesBF->addTerm(u->x(), tau1->div());
stokesBF->addTerm((1.0/mu) * sigma1, tau1);
stokesBF->addTerm(-u1_hat, tau1->dot_normal());

stokesBF->addTerm(u->y(), tau2->div());
stokesBF->addTerm((1.0/mu) * sigma2, tau2);
stokesBF->addTerm(-u2_hat, tau2->dot_normal());
```

This completes the definition of the bilinear form. We will see how to use this bilinear form to solve the lid-driven cavity flow problem in Section 3.4 below.

3.3 The Cavity Flow Problem

A classic challenge problem for Stokes flow is the lid-driven cavity flow problem. A viscous fluid is confined in a square cavity with a lid; the lid moves with unit velocity. This induces a vorticular flow within the cavity;

there are sequences of vortices with alternating direction going into each corner (these are known as *Moffatt eddies* [12]). Now, the obvious boundary conditions for this problem—unit velocity at the lid, with zero velocity at the walls—will involve a discontinuity in the velocity at the top corners. This is non-physical, and mathematically places the solution outside of H^1. Physically, there will be some interpolation between the unit velocity at the lid and the zero velocity at the corners. In our approach to this problem, we perform a simple linear interpolation over a distance $\epsilon = \frac{1}{64}$, as shown in the schematic in Figure 3.1. In ultraweak DPG formulations, boundary

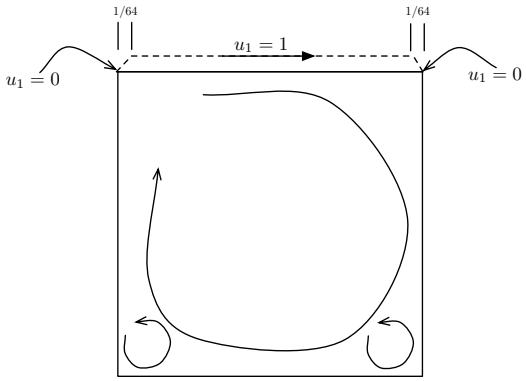


Figure 3.1: Lid-driven cavity flow schematic. The boundary conditions interpolate between $u_1 = 1$ at the lid and $u_1 = 0$ along the side walls. (Zero conditions are imposed on u_2 the lid and on all walls.)

conditions are generally imposed on the trace variables. Here, we impose boundary conditions on $\widehat{\boldsymbol{u}}$.

The cavity flow problem has a zero forcing function $\boldsymbol{f}$. As is generally true for incompressible flow problems, the Stokes equations only define the pressure up to a constant. Mathematically, the usual way to specify a unique pressure is to place $p \in L^2_{\text{avg}}(\Omega)$; that is, to require that pressure have zero average on the domain. Computationally, we may simply impose that the pressure is zero at some point in the domain; if desired, we can post-process to recover the zero-average solution.

3.4 Cavity Flow Implementation

3.4.1 Defining the Mesh

We would like to solve the cavity flow problem on an adaptive high-order mesh. We will start with a quartic 2×2 mesh, and perform h-refinements according to the DPG energy error measured. The `MeshFactory` class defines some convenient methods for rapidly specifying some simple meshes. One of these allows the definition of an axis-aligned rectilinear mesh. Camellia makes a distinction between the *mesh topology*, which defines the topology and the geometry of the mesh, and the *mesh*, which additionally defines polynomial discretizations for each variable on each element. These are implemented by the `MeshTopology` and `Mesh` classes, respectively. Here, we define the mesh topology for the cavity problem:

```
vector<double> dims = {1.0,1.0}; // domain dimensions
vector<int> meshDims = {2,2};  // 2x2 initial mesh
vector<double> x0 = {0.0,0.0}; // lower-left corner at origin

MeshTopologyPtr meshTopo;
meshTopo = MeshFactory::rectilinearMeshTopology(dims,
                                                meshDims, x0);
```

Next, we use the `meshTopo` to create a `Mesh` object with quartic field variables for the Stokes bilinear form. Because the field variables belong to L^2, this means that the H^1 order should be 5. We also need to specify the polynomial enrichment order to use for the DPG test space; a rule of thumb is to use the spatial dimension.[3]

```
int H1Order = 5;
int delta_k = 2;
MeshPtr mesh = MeshFactory::minRuleMesh(meshTopo,stokesBF,
                                        H1Order, delta_k);
```

The `minRuleMesh()` method produces a mesh that, on refinement, will use the standard finite element *minimum rule* to constrain the solution between adjacent elements that disagree on the fineness of the solution; the minimum rule essentially states that the element with the minimal discretization (in

[3]Note that the local costs can increase dramatically, especially in higher dimensions, for larger polynomial enrichments. Sometimes, good results may be obtained with smaller polynomial enrichments; we have generally had good luck with `delta_k` $= 1$ for Stokes in 2D and 3D, e.g.

terms of degrees of freedom) should constrain any other elements that share an interface with it.[4]

3.4.2 Boundary Conditions and Right-Hand Side

To specify our problem, we need to specify boundary conditions and the right-hand side of the equation. We instantiate a `BC` object which will store the boundary conditions:

```
BCPtr bc = BC::bc();
```

The boundary conditions are slightly involved because of the interpolation from the zero velocity at the wall to the unit velocity at the lid. We begin by defining a `Function` subclass that implements this interpolation along the lid at $y = 1$:

```
class LidVelocity : public SimpleFunction<double>
{
  double _eps; // interpolation width
public:
  LidVelocity(double eps)
  {
    _eps = eps;
  }
  double value(double x, double y)
  {
    if (abs(x) < _eps)
    {
      return x / _eps;          // top left
    }
    else if (abs(1.0-x) < _eps)
    {
      return (1.0-x) / _eps;    // top right
    }
    else
    {
      return 1;                 // top middle
    }
  }
};
```

[4] In an earlier, 2D-only version of the code, Camellia employed the maximum rule. Maximum rule support for 2D remains available in the present version of the code, but it should be considered deprecated.

This class definition could be placed in a separate `.h` file, or above the `main()` method in the driver file. Here, our `LidVelocity` subclass is a subclass of `SimpleFunction`, itself a subclass of the main `Function` class. `SimpleFunction` allows the definition of functions that only depend on spatial coordinates (i.e., functions that do not depend on the mesh).

Now, in the main driver, we can instantiate our subclass as follows:

```
double eps = 1.0 / 64.0;
FunctionPtr lidVelocity = Teuchos::rcp(new LidVelocity(eps));
```

Now, we wish to specify the boundary region along which the boundary conditions should be applied—in this case, the region of the boundary where $y = 1$. Camellia provides two mechanisms for this—the first, using *tags*, is intended primarily for meshes imported from an external source, typically with complicated geometry. The second, which we use here, is the `SpatialFilter` class, which specifies a region of space that should be matched. Spatial filters are always applied to the boundary of the mesh. `SpatialFilter` implements an overload of the `!` (not) operator, so that we can define the lid and wall filters as follows:

```
SpatialFilterPtr lid  = SpatialFilter::matchingY(1.0);
SpatialFilterPtr wall = !lid; // not lid --> wall
```

We can then define Dirichlet conditions on the x-velocity trace:

```
FunctionPtr zero = Function::constant(0); // for wall velocity
bc->addDirichlet(u1_hat, lid, lidVelocity);
bc->addDirichlet(u1_hat, wall, zero);
```

For the y-velocity trace conditions, we want to impose a zero condition along both the lid and walls; we take advantage of another overload, of the `|` (or) operator, which allows us to write `lid | wall` to specify that we want the boundary condition to apply at both the lid and walls:

```
bc->addDirichlet(u2_hat, lid | wall, zero);
```

We also need to specify the condition on the pressure. We could implement a zero-mean constraint as follows:

```
bc->addZeroMeanConstraint(p);
```

Alternately, we can impose a zero value on the pressure at the center of the mesh (note that the following depends on the fact that we have a vertex at the center of the mesh—recall that we defined a 2×2 mesh):

```
vector<double> center = {0.5,0.5};
double p_value = 0;
bc->addSpatialPointBC(p->ID(), p_value, center);
```

We are now ready to define the right-hand side. Since this is zero, the following will suffice:

```
RHSPtr rhs = RHS::rhs();
```

If we had `FunctionPtr` objects `f1` and `f2` that implemented $\boldsymbol{f}$ from equation (3.1.1), we would write the following:

```
rhs->addTerm(f1 * v1);
rhs->addTerm(f2 * v2);
```

3.4.3 The Test Space Inner Product

To define our DPG method, we need to specify an inner product on the test space. Frequently, a good choice for ultraweak DPG is the *graph norm* on the test space, which effectively minimizes the solution residual in the L^2 norm. The `IP` class implements the inner product, and the `BF` class provides a `graphNorm()` method that returns the graph norm on the test space. Thus we define our inner product:

```
IPPtr ip = stokesBF->graphNorm();
```

3.4.4 Solution and Visualization

Now, we may define the `Solution` object:

```
SolutionPtr soln = Solution::solution(stokesBF,mesh,bc,rhs,ip);
```

We can now solve the problem by calling:

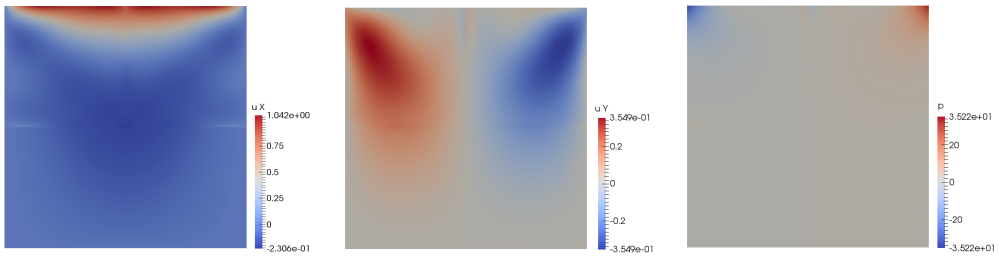
```
soln->solve();
```

We can export the solution in an HDF5 format suitable for viewing in ParaView by using an `HDF5Exporter` as follows:

```
int refNumber = 0;
HDF5Exporter exporter(mesh,"stokes-cavity-flow");
int numSubdivisions = 30; // coarse mesh -> more subdivisions
exporter.exportSolution(soln, refNumber, numSubdivisions);
```

The code above will place a `stokes-cavity-flow` directory in the working directory. This directory contains two files, `stokes-cavity-flow-field.xmf` and `stokes-cavity-flow-trace.xmf`, corresponding to the field and trace variables of the solution. The `numSubdivisions` argument controls how many points are plotted along an edge—for quadrilateral elements, approximately the square of this number of points will be plotted. Thus 30 is a large number to use, but for this coarse a quartic mesh, 30 is about the minimum to avoid visual artifacts. The `refNumber` argument is provided as the time value to the `HDF5Exporter`; this will allow us to store multiple refinements in a single visualization.

Opening the field `xmf` file with ParaView, we can plot the x and y component of the velocity field $\boldsymbol{u}$, as well as the pressure p; these are shown in Figure 3.2. Two things are worth noting regarding the solution: first, there are visible discontinuities between elements; second, and perhaps not obvious from the plots, the boundary conditions are not fully resolved by the mesh. Because $\epsilon = \frac{1}{64}$, we will need to perform five refinements into the upper corners before the mesh is able to capture the boundary conditions exactly.



(a) u_1 (b) u_2 (c) p

Figure 3.2: Lid-driven cavity flow: u_1, u_2, p solutions on initial 2×2, quartic mesh.

3.4.5 Adaptive Mesh Refinements

DPG provides us with a way to measure the energy norm of the residual on each element; the energy norm is the one in which DPG minimizes the residual. This is therefore a very natural quantity to use as an error indicator in driving adaptive refinements. The `RefinementStrategy`

class uses the energy norm as its default error indicator. We initialize a `RefinementStrategy` object as follows:

```
double threshold = 0.2; // relative energy error threshold
RefinementStrategyPtr refStrategy =
 RefinementStrategy::energyErrorRefinementStrategy(soln,
                                                    threshold);
```

During a refinement, this `refStrategy` will compute the energy error on each element, determine the maximum error $e_{\max}$, and then refine any element that has error greater than $0.2 * e_{\max}$. The following loop will refine, solve, and store the visualization data for each refinement:

```
int numRefinements = 8;
bool printToConsole = true;
for (refNumber = 1; refNumber < numRefinements; refNumber++)
{
  refStrategy->refine(printToConsole);
  soln->solve();
  exporter.exportSolution(soln, refNumber, numSubdivisions);
}
// report final energy error:
double energyError = soln->energyErrorTotal();
cout << "Final energy error: " << energyError << endl;
```

This produces the mesh shown in Figure 3.3;[5] the solution after 8 refinements can be seen in Figure 3.4.

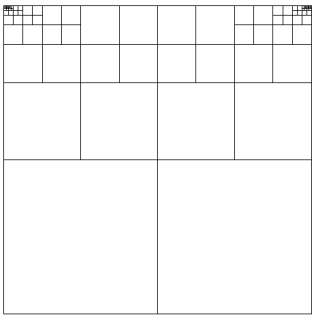
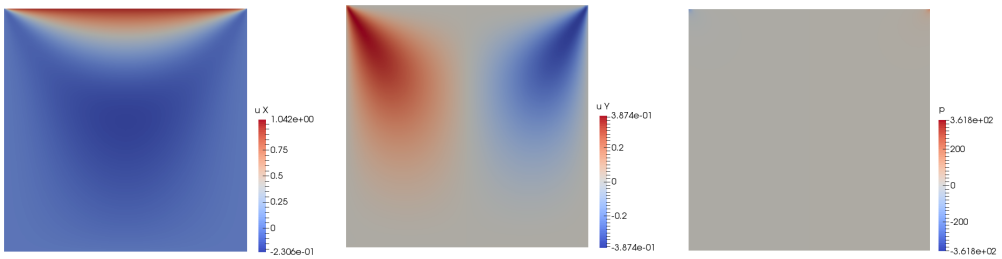


Figure 3.3: Lid-driven cavity flow: quartic mesh after 8 adaptive refinements.

[5]To produce a mesh plot like this one, one can open the trace `xmf` file in ParaView, and plot a black Solid Color.



(a) u_1 (b) u_2 (c) p

Figure 3.4: Lid-driven cavity flow: u_1, u_2, p solutions on quartic mesh after 8 adaptive refinements.

3.5 Stokes Cavity Flow Driver

The complete driver for the Stokes adaptive cavity flow driver is included in the Camellia distribution, under `manual-examples`. It is also listed below.

```
//
//   2016 UChicago Argonne.  For licensing details, see LICENSE-Camellia in the licenses directory.
//

#include "Camellia.h"

using namespace Camellia;
using namespace std;

class LidVelocity : public SimpleFunction<double>
{
  double _eps; // interpolation width
public:
  LidVelocity(double eps)
  {
    _eps = eps;
  }
  double value(double x, double y)
  {
    if (abs(x) < _eps)
    {
      return x / _eps;         // top left
    }
    else if (abs(1.0-x) < _eps)
    {
      return (1.0-x) / _eps;   // top right
    }
    else
    {
      return 1;                // top middle
    }
  }
};

int main(int argc, char *argv[])
{
  Teuchos::GlobalMPISession mpiSession(&argc, &argv); // initialize MPI

  VarFactoryPtr vf = VarFactory::varFactory();

  // field variables:
  VarPtr u = vf->fieldVar("u", VECTOR_L2);
  VarPtr p = vf->fieldVar("p", L2);
  VarPtr sigma1 = vf->fieldVar("sigma_1", VECTOR_L2);
  VarPtr sigma2 = vf->fieldVar("sigma_2", VECTOR_L2);

  // trace and flux variables:
  VarPtr u1_hat = vf->traceVar("u1_hat", HGRAD);
  VarPtr u2_hat = vf->traceVar("u2_hat", HGRAD);
  VarPtr tn1_hat = vf->fluxVar("tn1_hat", L2);
  VarPtr tn2_hat = vf->fluxVar("tn2_hat", L2);

  // test variables:
  VarPtr q = vf->testVar("q", HGRAD);
  VarPtr v1 = vf->testVar("v_1", HGRAD);
  VarPtr v2 = vf->testVar("v_2", HGRAD);
  VarPtr tau1 = vf->testVar("tau_1", HDIV);
  VarPtr tau2 = vf->testVar("tau_2", HDIV);

  // create BF object:
  BFPtr stokesBF = BF::bf(vf);

  // get a normal function (will be useful in a moment):
  FunctionPtr n = Function::normal();
  double mu = 1.0; // unit viscosity

  // add terms for v1:
  stokesBF->addTerm(sigma1, v1->grad());
  stokesBF->addTerm(-p, v1->dx());
```

```
  stokesBF->addTerm(-tn1_hat, v1);

  // add terms for v2:
  stokesBF->addTerm(sigma2, v2->grad());
  stokesBF->addTerm(-p, v2->dy());
  stokesBF->addTerm(-tn2_hat, v2);

  // add terms for q:
  stokesBF->addTerm(u, q->grad());
  stokesBF->addTerm(-u1_hat * n->x() - u2_hat * n->y(), q);

  // add terms for tau1:
  stokesBF->addTerm(u->x(), tau1->div());
  stokesBF->addTerm((1.0/mu) * sigma1, tau1);
  stokesBF->addTerm(-u1_hat, tau1->dot_normal());

  // add terms for tau2:
  stokesBF->addTerm(u->y(), tau2->div());
  stokesBF->addTerm((1.0/mu) * sigma2, tau2);
  stokesBF->addTerm(-u2_hat, tau2->dot_normal());

  vector<double> dims = {1.0,1.0}; // domain dimensions
  vector<int> meshDims = {2,2};  // 2x2 initial mesh
  vector<double> x0 = {0.0,0.0}; // lower-left corner at origin

  MeshTopologyPtr meshTopo;
  meshTopo = MeshFactory::rectilinearMeshTopology(dims, meshDims, x0);
  int H1Order = 5;
  int delta_k = 2;
  MeshPtr mesh = MeshFactory::minRuleMesh(meshTopo, stokesBF, H1Order, delta_k);
  BCPtr bc = BC::bc();

  double eps = 1.0 / 64.0;
  FunctionPtr lidVelocity = Teuchos::rcp(new LidVelocity(eps));

  SpatialFilterPtr lid  = SpatialFilter::matchingY(1.0);
  SpatialFilterPtr wall = !lid; // not lid --> wall

  FunctionPtr zero = Function::constant(0); // for wall velocity
  bc->addDirichlet(u1_hat, lid, lidVelocity);
  bc->addDirichlet(u1_hat, wall, zero);

  bc->addDirichlet(u2_hat, lid | wall, zero);

  vector<double> center = {0.5,0.5};
  double p_value = 0;
  bc->addSpatialPointBC(p->ID(), p_value, center);

  RHSPtr rhs = RHS::rhs();

  IPPtr ip = stokesBF->graphNorm();
  SolutionPtr soln = Solution::solution(stokesBF,mesh,bc,rhs,ip);

  soln->solve();

  int refNumber = 0;
  HDF5Exporter exporter(mesh,"stokes-cavity-flow");
  int numSubdivisions = 30; // coarse mesh -> more subdivisions
  exporter.exportSolution(soln, refNumber, numSubdivisions);

  double threshold = 0.2; // relative energy error threshold
  RefinementStrategyPtr refStrategy = RefinementStrategy::energyErrorRefinementStrategy(soln,
                                                                                         threshold);
  int numRefinements = 8;
  bool printToConsole = true;
  for (refNumber = 1; refNumber < numRefinements; refNumber++)
  {
    refStrategy->refine(printToConsole);
    soln->solve();
    exporter.exportSolution(soln, refNumber, numSubdivisions);
  }
  // report final energy error:
  double energyError = soln->energyErrorTotal();
  cout << "Final energy error: " << energyError << endl;

  return 0;
}
```

Chapter 4

Navier-Stokes Cavity Flow Using DPG

In this chapter, we solve the cavity flow problem introduced in Chapter 3, now using the incompressible Navier-Stokes equations.[1] These are nonlinear equations. When dealing with nonlinear problems, the usual DPG approach is to apply the DPG methodology to the linearized equations, and perform Newton steps on these until a converged solution is found, and this is what we support in Camellia. The equations are essentially the same as the Stokes equations, except there is a nonlinear convective term $\boldsymbol{u} \cdot \nabla \boldsymbol{u}$:

$$\begin{aligned} -\frac{1}{\mathrm{Re}} \Delta \boldsymbol{u} + \nabla p &= \boldsymbol{f} - \boldsymbol{u} \cdot \nabla \boldsymbol{u} && \text{in } \Omega, \\ \nabla \cdot \boldsymbol{u} &= 0 && \text{in } \Omega. \end{aligned}$$

We identify μ from the Stokes equations with $\frac{1}{\mathrm{Re}}$, where Re is the *Reynolds number*, a non-dimensional parameter that characterizes incompressible flow.

4.1 Ultraweak Formulation for Navier-Stokes

Introducing $\boldsymbol{\sigma} = \frac{1}{\mathrm{Re}} \nabla \boldsymbol{u}$, we have the following system:

[1] For a fuller exploration of the DPG formulation for Navier-Stokes that we derive here, see Roberts et al. [17].

$$
\begin{aligned}
-\nabla\cdot\boldsymbol{\sigma}+\nabla p+\operatorname{Re}\boldsymbol{u}\cdot\boldsymbol{\sigma}&=\boldsymbol{f} && \text{in } \Omega,\\
\operatorname{Re}\boldsymbol{\sigma}-\nabla\boldsymbol{u}&=0 && \text{in } \Omega,\\
\nabla\cdot\boldsymbol{u}&=0 && \text{in } \Omega,\\
\boldsymbol{u}&=\boldsymbol{u}_D && \text{on } \partial\Omega.
\end{aligned}
$$

Testing as before and integrating by parts, we get the following nonlinear formulation:

$$
\begin{aligned}
b_{\text{Navier-Stokes}}(u,v) \overset{\text{def}}{=} &(\boldsymbol{\sigma}-p\boldsymbol{I},\nabla\boldsymbol{v})_{\Omega_h}+(\operatorname{Re}\boldsymbol{u}\cdot\boldsymbol{\sigma},\boldsymbol{v})_{\Omega_h}-\langle\widehat{\boldsymbol{t}}_n,\boldsymbol{v}\rangle_{\partial\Omega_h}\\
&+(\boldsymbol{u},\nabla q)_{\Omega_h}-\langle\widehat{\boldsymbol{u}}\cdot\boldsymbol{n},q\rangle_{\partial\Omega_h} \qquad (4.1.1)\\
&+(\operatorname{Re}\boldsymbol{\sigma},\boldsymbol{\tau})_{\Omega_h}+(\boldsymbol{u},\nabla\cdot\boldsymbol{\tau})_{\Omega_h}-\langle\widehat{\boldsymbol{u}},\boldsymbol{\tau}\boldsymbol{n}\rangle_{\partial\Omega_h}=(\boldsymbol{f},\boldsymbol{v})_{\Omega_h},
\end{aligned}
$$

which may be written in terms of the Stokes formulation:

$$
b_{\text{Navier-Stokes}}(u,v)=b_{\text{Stokes}}(u,v)+(\operatorname{Re}\boldsymbol{u}\cdot\boldsymbol{\sigma},\boldsymbol{v})_{\Omega_h}=l(\boldsymbol{v}).
$$

Linearizing about a background flow $u=(\boldsymbol{u},p,\boldsymbol{\sigma},\widehat{\boldsymbol{u}},\widehat{\boldsymbol{t}}_n)$ with solution increment $\Delta u=(\Delta\boldsymbol{u},\Delta p,\Delta\boldsymbol{\sigma},\Delta\widehat{\boldsymbol{u}},\Delta\widehat{\boldsymbol{t}}_n)$, we have:

$$
b_{\text{Stokes}}(u+\Delta u,v)+(\operatorname{Re}(\boldsymbol{u}+\Delta\boldsymbol{u})\cdot(\boldsymbol{\sigma}+\Delta\boldsymbol{\sigma}),\boldsymbol{v})_{\Omega_h}=l(\boldsymbol{v}).
$$

Dropping the the high-order term $(\Delta\boldsymbol{u}\cdot\Delta\boldsymbol{\sigma},\boldsymbol{v})_{\Omega_h}$ and moving the known terms to the right-hand side, we obtain:

$$
\begin{aligned}
b_{\text{Stokes}}(\Delta u,v)+&(\operatorname{Re}(\boldsymbol{u}\cdot\Delta\boldsymbol{\sigma}+\Delta\boldsymbol{u}\cdot\boldsymbol{\sigma}),\boldsymbol{v})_{\Omega_h} \qquad (4.1.2)\\
&=l(\boldsymbol{v})-b_{\text{Stokes}}(u,v)-(\operatorname{Re}\boldsymbol{u}\cdot\boldsymbol{\sigma},\boldsymbol{v})_{\Omega_h}.
\end{aligned}
$$

Note that the Navier-Stokes variables in this formulation are identical to those in the Stokes formulation; thus we may use exactly the same code to define these as in Chapter 3. As suggested by (4.1.2), we will first set up a Stokes bilinear form, and then define our linearized Navier-Stokes formulation in terms of this. The below code, defining `stokesBF`, is nearly identical to that in the last chapter; the only difference is the use of `Re` in place of `mu`:

```
double Re = 1e2;

// add terms for v1:
stokesBF->addTerm(sigma1, v1->grad());
```

```
stokesBF->addTerm(-p, v1->dx());
stokesBF->addTerm(-tn1_hat, v1);

// add terms for v2:
stokesBF->addTerm(sigma2, v2->grad());
stokesBF->addTerm(-p, v2->dy());
stokesBF->addTerm(-tn2_hat, v2);

// add terms for q:
stokesBF->addTerm(u, q->grad());
stokesBF->addTerm(-u1_hat * n->x() - u2_hat * n->y(), q);

// add terms for tau1:
stokesBF->addTerm(sigma1,tau1);
stokesBF->addTerm(u->x(), tau1->div());
stokesBF->addTerm(Re * sigma1, tau1);
stokesBF->addTerm(-u1_hat, tau1->dot_normal());

// add terms for tau2:
stokesBF->addTerm(sigma2,tau2);
stokesBF->addTerm(u->y(), tau2->div());
stokesBF->addTerm(Re * sigma2, tau2);
stokesBF->addTerm(-u2_hat, tau2->dot_normal());
```

We define the mesh in exactly the same way as in Chapter 3:

```
MeshTopologyPtr meshTopo;
meshTopo = MeshFactory::rectilinearMeshTopology(dims,
                                                meshDims, x0);
int H1Order = 5;
int delta_k = 2;
MeshPtr mesh = MeshFactory::minRuleMesh(meshTopo, stokesBF,
                                        H1Order, delta_k);
```

Note that we use the `stokesBF` to initialize the mesh, even though we are actually interested in solving the Navier-Stokes equations. This is okay because the two formulations will share a `VarFactory`—i.e., they have exactly the same variables, enumerated in exactly same way. Similarly, we can now define a background flow solution using `stokesBF`:

```
SolutionPtr backFlow = Solution::solution(stokesBF,mesh);
```

Here, we omit the `bc`, `rhs`, and `ip` arguments (which are then assigned null values), because this `Solution` will only be used for storing solution coefficients, not for actually solving. When we refine, we would like to project

the refined cell's background flow onto its children; to ensure that this will happen automatically, we can register the solution with the mesh as follows:

```
mesh->registerSolution(backFlow);
```

We are now ready to define our background flow functions; we will only need $\boldsymbol{u}$ and $\boldsymbol{\sigma}$:

```
FunctionPtr u_prev = Function::solution(u,backFlow);
FunctionPtr sigma1_prev = Function::solution(sigma1,backFlow);
FunctionPtr sigma2_prev = Function::solution(sigma2,backFlow);
```

Now, we create our Navier-Stokes bilinear form, and add terms corresponding to $-\left(\mathrm{Re}\left(\boldsymbol{u}\cdot\Delta\boldsymbol{\sigma}+\Delta\boldsymbol{u}\cdot\boldsymbol{\sigma}\right),\boldsymbol{v}\right)_{\Omega_h}$:

```
BFPtr nsBF = stokesBF->copy();

nsBF->addTerm(Re * u_prev * sigma1, v1);
nsBF->addTerm(Re * u_prev * sigma2, v2);

nsBF->addTerm(Re * sigma1_prev * u, v1);
nsBF->addTerm(Re * sigma2_prev * u, v2);
```

The call to `stokesBF->copy()` creates a copy of the object that `stokesBF` points to, allowing us to make modifications that do not affect the original `stokesBF`. (We will now use the original `stokesBF` to modify the right-hand side.)

4.2 Adjusting the RHS for Navier-Stokes Linearization

If we assume that we have the right-hand side object `rhs` set up to correspond to our forcing function $\boldsymbol{f}$ (in the case of cavity flow $\boldsymbol{f}=\boldsymbol{0}$), then we can adjust it as follows:

```
LinearTermPtr stokesTerm = stokesBF->testFunctional(backFlow);
rhs->addTerm(-stokesTerm);
```

The `BF::testFunctional(soln)` method evaluates[2] a bilinear form at a given point in the trial space—a solution `soln`—producing a functional on

[2]Note that this does not mean that `testFunctional()` computes much of anything at the time that it is called; instead, a `LinearTermPtr` is created which has references to the `soln` object; this will be used when the linear term is evaluated in an integral computation.

the test space. In other words, this method "fills in" the solution `soln` for the trial space variables in the bilinear form. Finally, we adjust `rhs` with a term corresponding to $(-\mathrm{Re}\,\boldsymbol{u}\cdot\boldsymbol{\sigma},\boldsymbol{v})_{\Omega_h}$:

```
rhs->addTerm(-Re * u_prev * sigma1_prev * v1);
rhs->addTerm(-Re * u_prev * sigma2_prev * v2);
```

4.3 Test Space Inner Product

We now create our inner product, the graph norm corresponding to our Navier-Stokes formulation:

```
IPPtr ip = nsBF->graphNorm();
```

Note that because `nsBF` depends on the previous solution (`backFlow`), `ip` does as well. (As the background flow is updated, the norm on the test space will change in such a way as to preserve the equivalence of the DPG energy norm with the L^2 norm.)

4.4 Boundary Conditions

There is a subtlety involving the boundary conditions: in each Newton step, we will be solving for the solution *increment*, so we need to write boundary conditions accordingly. For the boundary conditions where zero is imposed no modification is necessary, but anywhere else, to impose a function `bcFxn` on trace `var` for a boundary region matched by a `SpatialFilter` `sf` we need code of the form:

```
FunctionPtr var_prev = Function::solution(var, backFlow);
bc->addDirichlet(var, sf, bcFxn - var_prev);
```

In the particular case of the cavity flow boundary conditions, we replace the code defining $\widehat{u}_1$ at the lid with the following:

```
FunctionPtr u1_hat_prev = Function::solution(u1_hat,backFlow);
bc->addDirichlet(u1_hat, lid, lidVelocity - u1_hat_prev);
```

Otherwise, we define `bc` just as in Chapter 3.

4.5 The Navier-Stokes Solve and the Newton Stopping Criterion

We are now ready to create the Navier-Stokes solution object:

```
SolutionPtr soln = Solution::solution(nsBF,mesh,bc,rhs,ip);
```

Now, when we call `soln->solve()`, this will solve for the solution increment Δu, with background flow u as stored in `backFlow`. For cavity flow, we will be happy with a zero initial guess; if we wanted something else, we could call `backFlow->projectOntoMesh()`; this takes as argument a C++ `map` with variable IDs as keys, and `FunctionPtr`s as values. After we solve for the increment, we want to add it to the background flow; we want to repeatedly do this until some stopping criterion is reached.

One simple stopping criterion is to stop when the combined L^2 norm of the field variables $\boldsymbol{u}, p, \boldsymbol{\sigma}$ in the solution increment `soln` gets below a threshold. We will consider an appropriate threshold in a moment; for now, we set up a `Function` that will allow us to measure this value:

```
FunctionPtr u_incr = Function::solution(u,soln);
FunctionPtr p_incr = Function::solution(p,soln);
FunctionPtr sigma1_incr = Function::solution(sigma1,soln);
FunctionPtr sigma2_incr = Function::solution(sigma2,soln);
FunctionPtr l2_squared = u_incr * u_incr + p_incr * p_incr
    + sigma1_incr * sigma1_incr + sigma2_incr * sigma2_incr;
```

Now, we set up a loop for the Newton iteration, with a starting threshold of 10^{-3}:

```
double newtonThreshold = 1e-3;
double l2_incr = 0;
do
{
  soln->solve();
  // add increment with unit weight:
  backFlow->addSolution(soln, 1.0);
  l2_incr = sqrt( l2_squared->integrate(mesh) );
} while (l2_incr > newtonThreshold);
```

4.6 A Dynamic Newton Threshold

We can enclose the above loop in another loop that performs adaptive mesh refinements. However, what we will find is that a fixed threshold for the stopping criterion is not ideal: if the threshold is too small, we may never converge on a coarse mesh; if it is too large, we may not take enough Newton steps to resolve the solution on fine meshes. For this reason, we often employ a dynamic threshold. One approach is to take threshold $\epsilon_0 = 10^{-3}$ for the initial mesh. Then, on the nth refinement we measure the DPG energy error of the solution increment, e^n, defined as

$$e^n = \|\Delta u^n - \Delta u_h^n\|_E = \|l(\cdot) - b(\Delta u_h^n, \cdot)\|_{V_h'} .$$

The rightmost expression is the energy norm of the solution residual, precisely what `RefStrategy` computes in the course of adaptive refinements. We would like the threshold to scale with the magnitude of the background flow; we therefore define the *relative* error as

$$e_{\text{rel}}^n = \frac{e^n}{\|(\boldsymbol{u}^n, p^n, \boldsymbol{\sigma}^n)\|_{L^2}}.$$

We now define the stopping criterion threshold for the nth refinement as $\epsilon_n = e_{\text{rel}}^{n-1}\epsilon_0$. As we refine, the DPG energy error e^n diminishes, and the tolerance for the Newton steps will grow tighter. The following code implements the refinement loop, complete with a dynamically updated threshold:

```
FunctionPtr p_prev = Function::solution(p, backFlow);
FunctionPtr l2_backFlow_squared = u_prev * u_prev
 + p_prev * p_prev
 + sigma1_prev * sigma1_prev + sigma2_prev * sigma2_prev;

double newtonThreshold = 1e-3;
double energyThreshold = 0.2;
RefinementStrategyPtr refStrategy =
RefinementStrategy::energyErrorRefinementStrategy(soln,
                                                   energyThreshold);
int numRefinements = 8;
bool printToConsole = true;
double energyError, l2_soln;
for (refNumber = 0; refNumber <= numRefinements; refNumber++)
{
  do
  {
```

```
    soln->solve();
    l2_incr = sqrt( l2_squared->integrate(mesh) );
    // add increment with unit weight:
    backFlow->addSolution(soln, 1.0);
  }
  while (l2_incr > newtonThreshold);

  refStrategy->refine(printToConsole);
  l2_soln = sqrt(l2_backFlow_squared->integrate(mesh));
  // update threshold:
  energyError = refStrategy->getEnergyError(refNumber);
  newtonThreshold = 1e-3 * energyError / l2_soln;
}
```

The mesh and solution after 8 refinements are shown in Figures 4.1 and 4.2, respectively. Note that, in contrast to the Stokes solutions from Chapter 3, there is a slight asymmetry in both the solution and the adaptive refinements.

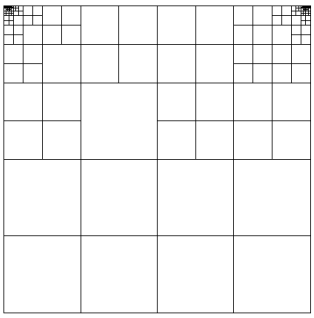
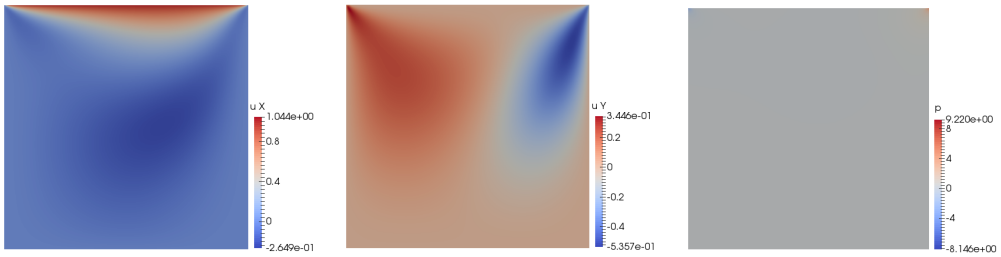


Figure 4.1: Lid-driven cavity flow for Re = 100: quartic mesh after 8 adaptive refinements.



(a) u_1 (b) u_2 (c) p

Figure 4.2: Lid-driven cavity flow for Re = 100: u_1, u_2, p solutions on quartic mesh after 8 adaptive refinements.

4.7 Navier-Stokes Cavity Flow Driver

The complete driver for the Navier-Stokes adaptive cavity flow driver is included with the Camellia distribution, under `manual-examples`. It is also listed below.

```
//
//   2016 UChicago Argonne.  For licensing details, see LICENSE-Camellia in the licenses directory.
//

#include "Camellia.h"

using namespace Camellia;
using namespace std;

class LidVelocity : public SimpleFunction<double>
{
  double _eps; // interpolation width
public:
  LidVelocity(double eps)
  {
    _eps = eps;
  }
  double value(double x, double y)
  {
    if (abs(x) < _eps)
    {
      return x / _eps;         // top left
    }
    else if (abs(1.0-x) < _eps)
    {
      return (1.0-x) / _eps;   // top right
    }
    else
    {
      return 1;                // top middle
    }
  }
};

int main(int argc, char *argv[])
{
  Teuchos::GlobalMPISession mpiSession(&argc, &argv); // initialize MPI
  int rank = MPIWrapper::CommWorld()->MyPID();

  VarFactoryPtr vf = VarFactory::varFactory();

  // field variables:
  VarPtr u = vf->fieldVar("u", VECTOR_L2);
  VarPtr p = vf->fieldVar("p", L2);
  VarPtr sigma1 = vf->fieldVar("sigma_1", VECTOR_L2);
  VarPtr sigma2 = vf->fieldVar("sigma_2", VECTOR_L2);

  // trace and flux variables:
  VarPtr u1_hat = vf->traceVar("u1_hat", HGRAD);
  VarPtr u2_hat = vf->traceVar("u2_hat", HGRAD);
  VarPtr tn1_hat = vf->fluxVar("tn1_hat", L2);
  VarPtr tn2_hat = vf->fluxVar("tn2_hat", L2);

  // test variables:
  VarPtr q = vf->testVar("q", HGRAD);
  VarPtr v1 = vf->testVar("v_1", HGRAD);
  VarPtr v2 = vf->testVar("v_2", HGRAD);
  VarPtr tau1 = vf->testVar("tau_1", HDIV);
  VarPtr tau2 = vf->testVar("tau_2", HDIV);

  // create Stokes BF object:
  BFPtr stokesBF = BF::bf(vf);

  // get a normal function (will be useful in a moment):
  FunctionPtr n = Function::normal();
  double Re = 1e2;
```

```
// add terms for v1:
stokesBF->addTerm(sigma1, v1->grad());
stokesBF->addTerm(-p, v1->dx());
stokesBF->addTerm(-tn1_hat, v1);

// add terms for v2:
stokesBF->addTerm(sigma2, v2->grad());
stokesBF->addTerm(-p, v2->dy());
stokesBF->addTerm(-tn2_hat, v2);

// add terms for q:
stokesBF->addTerm(-u, q->grad());
stokesBF->addTerm(u1_hat * n->x() + u2_hat * n->y(), q);

// add terms for tau1:
stokesBF->addTerm(u->x(), tau1->div());
stokesBF->addTerm(Re * sigma1, tau1);
stokesBF->addTerm(-u1_hat, tau1->dot_normal());

// add terms for tau2:
stokesBF->addTerm(u->y(), tau2->div());
stokesBF->addTerm(Re * sigma2, tau2);
stokesBF->addTerm(-u2_hat, tau2->dot_normal());

vector<double> dims = {1.0,1.0}; // domain dimensions
vector<int> meshDims = {2,2};  // 2x2 initial mesh
vector<double> x0 = {0.0,0.0}; // lower-left corner at origin

MeshTopologyPtr meshTopo;
meshTopo = MeshFactory::rectilinearMeshTopology(dims, meshDims, x0);
int H1Order = 5;
int delta_k = 2;
MeshPtr mesh = MeshFactory::minRuleMesh(meshTopo, stokesBF, H1Order, delta_k);

SolutionPtr backFlow = Solution::solution(stokesBF,mesh);
mesh->registerSolution(backFlow);

FunctionPtr u_prev = Function::solution(u,backFlow);
FunctionPtr sigma1_prev = Function::solution(sigma1,backFlow);
FunctionPtr sigma2_prev = Function::solution(sigma2,backFlow);

BFPtr nsBF = stokesBF->copy();
nsBF->addTerm(Re * u_prev * sigma1, v1);
nsBF->addTerm(Re * u_prev * sigma2, v2);

nsBF->addTerm(Re * sigma1_prev * u, v1);
nsBF->addTerm(Re * sigma2_prev * u, v2);

double eps = 1.0 / 64.0;
FunctionPtr lidVelocity = Teuchos::rcp(new LidVelocity(eps));

SpatialFilterPtr lid  = SpatialFilter::matchingY(1.0);
SpatialFilterPtr wall = !lid; // not lid --> wall

FunctionPtr zero = Function::constant(0); // for wall velocity

// for the non-zero velocity BC, need to impose on the
// *difference* between previously imposed velocity and
// the velocity we want (because we'll accumulate, and
// also because on the coarse meshes we won't get
// the BCs exactly).

FunctionPtr u1_hat_prev = Function::solution(u1_hat, backFlow);

BCPtr bc = BC::bc();
bc->addDirichlet(u1_hat, lid, lidVelocity - u1_hat_prev);
bc->addDirichlet(u1_hat, wall, zero);
bc->addDirichlet(u2_hat, lid | wall, zero);

RHSPtr rhs = RHS::rhs();
LinearTermPtr stokesTerm = stokesBF->testFunctional(backFlow);
rhs->addTerm(-stokesTerm);
rhs->addTerm(-Re * u_prev * sigma1_prev * v1);
rhs->addTerm(-Re * u_prev * sigma2_prev * v2);

vector<double> center = {0.5,0.5};
double p_value = 0;
```

```
  bc->addSpatialPointBC(p->ID(), p_value, center);

  IPPtr ip = nsBF->graphNorm();
  SolutionPtr soln = Solution::solution(nsBF,mesh,bc,rhs,ip);

  FunctionPtr u_incr = Function::solution(u,soln);
  FunctionPtr p_incr = Function::solution(p,soln);
  FunctionPtr sigma1_incr = Function::solution(sigma1,soln);
  FunctionPtr sigma2_incr = Function::solution(sigma2,soln);
  FunctionPtr l2_squared = u_incr * u_incr + p_incr * p_incr
    + sigma1_incr * sigma1_incr + sigma2_incr * sigma2_incr;

  FunctionPtr p_prev = Function::solution(p, backFlow);
  FunctionPtr l2_backFlow_squared = u_prev * u_prev + p_prev * p_prev
    + sigma1_prev * sigma1_prev + sigma2_prev * sigma2_prev;

  double l2_incr = 0;

  int refNumber = 0;
  HDF5Exporter exporter(mesh,"navier-stokes-cavity");
  int numSubdivisions = 30; // coarse mesh -> more subdivisions

  double newtonThreshold = 1e-3;
  double energyThreshold = 0.2;
  RefinementStrategyPtr refStrategy =
  RefinementStrategy::energyErrorRefinementStrategy(soln, energyThreshold);
  int numRefinements = 8;
  bool printToConsole = true;
  double energyError, l2_soln;
  for (refNumber = 0; refNumber <= numRefinements; refNumber++)
  {
    do
    {
      soln->solve();
      l2_incr = sqrt( l2_squared->integrate(mesh) );
      if (rank==0) cout << "L^2(increment): " << l2_incr << endl;
      // add increment with unit weight:
      backFlow->addSolution(soln, 1.0);
    }
    while (l2_incr > newtonThreshold);

    exporter.exportSolution(backFlow, refNumber, numSubdivisions);

    refStrategy->refine(printToConsole);
    energyError = refStrategy->getEnergyError(refNumber);
    l2_soln = sqrt(l2_backFlow_squared->integrate(mesh));
    // update threshold:
    newtonThreshold = 1e-3 * energyError / l2_soln;
    cout << "L^2(soln): " << l2_soln << endl;
    cout << "Newton threshold: " << newtonThreshold << endl;
  }
  energyError = soln->energyErrorTotal();
  cout << "Final energy error: " << energyError << endl;

  return 0;
}
```

Chapter 5

Formulations: Poisson, Stokes, and Navier-Stokes

In Chapters 3 and 4, we implemented drivers that included bilinear formulations for Stokes and Navier-Stokes, respectively. Generally, we prefer to avoid implementing such things in our drivers, for a few reasons. First, much of the time we will be interested in the same formulation in the context of several drivers. Second, the implementations are somewhat complex and error-prone—it is fairly easy to introduce a sign error, for example—and by having a single implementation that is reused, we can reduce the chances of bugs in our drivers. Finally, by having an implementation that is independent of the driver, we make it much simpler to test the implementation in our unit tests (we plan to address Camellia's approach to unit tests in Part 2 of this manual).

For these reasons, Camellia includes several "Formulation" classes which define some commonly used bilinear forms. In this chapter, we discuss three of these: `PoissonFormulation`, `StokesVGPFormulation`, and `NavierStokesVGPFormulation`. The latter two implement the same VGP formulations covered in Chapters 3 and 4, while the former implements the Poisson formulation derived in Chapter 2.

Some basic conventions adopted by the Formulation classes include:

- The `BFPtr` object corresponding to the bilinear form should be available via a `bf()` method.
- Each trial and test variable should be available through a method bearing its name.

- When there are subtleties involving appropriate application of boundary conditions—e.g., inflow conditions—the Formulation should include mechanisms by which the user can specify these simply.

5.1 The `PoissonFormulation` Class

Consider the Poisson equation

$$\Delta u = f.$$

In Section 2.3.1, we introduced an ultraweak variational formulation for this equation, in which we identified a new variable $\boldsymbol{\sigma}$ with ∇u:

$$\begin{aligned} b(\cdot,\cdot) = (-\boldsymbol{\sigma}, \nabla v) + \langle \widehat{\sigma}_n, v\rangle \\ + (\boldsymbol{\sigma}, \boldsymbol{\tau}) + (u, \nabla\cdot\boldsymbol{\tau}) - \langle \widehat{u}, \boldsymbol{\tau}\cdot\boldsymbol{n}\rangle = (f, v) \end{aligned}$$

This formulation is implemented by the `PoissonFormulation` class. The constructor for this class takes two required arguments: `spaceDim`, the spatial dimension of the mesh (values of 1, 2, and 3 are valid) and `conformingTraces`, a boolean governing the kind of continuity that is enforced on the H^1 trace $\widehat{u}$.[1] Because $\widehat{u}$ is the trace of an H^1 variable, a conforming discretization of $\widehat{u}$ in 3D will enforce continuity at vertices, edges, and faces; if `conformingTraces` is true, then such continuity will be enforced. If `conformingTraces` is false, then continuity will only be enforced on the sides (faces in 3D, edges in 2D). To create a `BF` object for a 3D ultraweak conforming Poisson formulation, one may write the following:

```
int spaceDim = 3;
bool conformingTraces = true;
PoissonFormulation form(spaceDim, conformingTraces);
BFPtr bf = form.bf();
```

An Example Problem

Suppose that we wish to solve the Poisson problem with unit forcing function $f = 1$ and homogeneous boundary conditions, on the unit cube $(0,1)^3$. We

[1] An optional third argument allows one to specify which Poisson formulation is desired; the default is the above ultraweak DPG formulation—other available formulations are the continuous Galerkin formulation and the primal DPG formulation.

define the `FunctionPtr f`, then request a corresponding `RHS` object from the Poisson formulation `form`:

```
FunctionPtr f = Function::constant(1.0);
RHSPtr rhs = form.rhs(f);
```

As we have seen, for DPG ultraweak formulations, we generally impose boundary conditions using the trace and flux variables—here we will want to impose $\widehat{u} = 0$ on the entire boundary of the domain. To set up the `BC` object, we need access to the variable $\widehat{u}$, which is available through the `u_hat()` method of `PoissonFormulation`. Thus we do the following:

```
BCPtr bc = BC::bc();
VarPtr u_hat = form.u_hat();
SpatialFilterPtr everywhere = SpatialFilter::allSpace();
bc->addDirichlet(u_hat, everywhere, Function::zero());
```

We can use `MeshFactory` to create a mesh corresponding to our domain; we start with a single element. We use an H^1 order of 3; this means that our L^2 field variables will be quadratic. We also use a polynomial enrichment of 3 for the test space; it has been proven for Poisson that this suffices to guarantee optimal convergence rates for the DPG method [11].

```
vector<int> elementCounts = {1,1,1}; // x,y,z directions
vector<double> domainDim = {1.0,1.0,1.0};
int H1Order = 3;
int delta_k = 3;
MeshPtr mesh = MeshFactory::rectilinearMesh(bf, domainDim,
                                            elementCounts,
                                            H1Order, delta_k);
```

The solution to our Poisson problem is quite smooth and isotropic, so that uniform refinements are a reasonable choice. We refine three times to produce an $8 \times 8 \times 8$ mesh:

```
int numRefs = 3;
for (int i=0; i<numRefs; i++)
{
  mesh->hRefine(mesh->getActiveCellIDsGlobal());
}
```

We now create a `Solution` and solve:

```
SolutionPtr soln = Solution::solution(bf,mesh,bc,rhs,
                                      bf->graphNorm());
soln->solve();
```

A 2D slice of the solution, taken on the center plane at $x = \frac{1}{2}$, can be found in Figure 5.1.

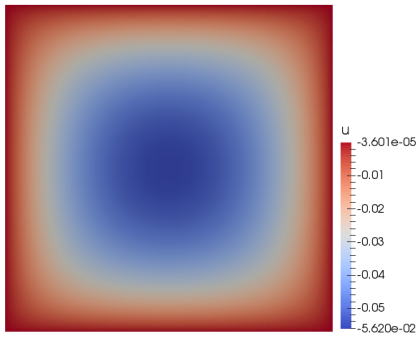


Figure 5.1: Poisson problem on unit cube with homogeneous BCs and unit forcing function: u solution at $x = \frac{1}{2}$ on quadratic $8 \times 8 \times 8$ mesh.

5.2 The `StokesVGPFormulation` Class

Camellia provides a `StokesVGPFormulation` class for the velocity-gradient-pressure ultraweak Stokes formulation discussed in Chapter 3. As with `PoissonFormulation`, both conforming and non-conforming traces are supported; `StokesVGPFormulation` also provides built-in features for boundary conditions, solving, and refinements. To implement a solver for the same cavity flow problem that we discussed in Chapter 3, we begin by creating the `StokesVGPFormulation` object:

```
int spaceDim = 2; // 3D also supported
double mu = 1.0;
bool conformingTraces = true;
StokesVGPFormulation form
   = StokesVGPFormulation::steadyFormulation(spaceDim, mu,
                                             conformingTraces);
```

We then create the `MeshTopology` exactly as before:

```
vector<double> dims = {1.0,1.0};// domain dimensions
vector<int> meshDims = {2,2};   // 2x2 initial mesh
vector<double> x0 = {0.0,0.0};  // lower-left corner at origin
MeshTopologyPtr meshTopo;
meshTopo = MeshFactory::rectilinearMeshTopology(dims,
```

```
                                                    meshDims, x0);
```

Next, we call `initializeSolution()`; `StokesVGPFormulation` will create a new `Solution` object internally. As before, we use a field polynomial order of 4, a test space enrichment $\Delta k = 2$, and a zero forcing function, $\boldsymbol{f}$.

```
int polyOrder = 4;
int delta_k = 2;
FunctionPtr f = Function::zero(1); // vector zero
form.initializeSolution(meshTopo, polyOrder, delta_k, f);
```

Assuming we have a `LidVelocity` class defined as in Chapter 3, we create a vector velocity function for the velocity at the lid:

```
double eps = 1.0 / 64.0;
FunctionPtr lidVelocity_x
     = Teuchos::rcp(new LidVelocity(eps));
FunctionPtr lidVelocity
     = Function::vectorize(lidVelocity_x, Function::zero());
```

We then define `SpatialFilter` objects corresponding to the lid and the walls, as we did before:

```
SpatialFilterPtr lid  = SpatialFilter::matchingY(1.0);
SpatialFilterPtr wall = !lid; // not lid --> wall
```

Now, we add wall conditions and the velocity condition at the lid. While there is not technically inflow at the lid, because the velocity is non-zero, we use `StokesVGPFormulation`'s `addInflowCondition()` method, which imposes Dirichlet conditions on $\widehat{\boldsymbol{u}}$. We also add the condition that the pressure is zero at $(0.5, 0.5)$:[2]

```
form.addInflowCondition(lid, lidVelocity);
form.addWallCondition(wall);
form.addPointPressureCondition({0.5,0.5});
```

We can then solve on the initial mesh by simply calling

```
form.solve();
```

We set up an exporter for visualization much as in Chapter 3:

```
int refNumber = 0;
SolutionPtr soln = form.solution();
HDF5Exporter exporter(soln->mesh(),"ch05-stokes-cavity-flow");
```

[2]Note that value imposition at a point requires that a vertex is defined at the point.

```
int numSubdivisions = 30; // coarse mesh -> more subdivisions
exporter.exportSolution(soln, refNumber, numSubdivisions);
```

`StokesVGPFormulation` also creates a `RefinementStrategy` when it creates the `Solution` object; here, we set the relative energy error threshold to 0.2, just as we had in Chapter 3—this means that we will refine all elements that have energy error greater than 20% of the maximum element energy error.

```
double threshold = 0.2; // relative energy error threshold
auto refStrategy = form.getRefinementStrategy();
refStrategy->setRelativeErrorThreshold(threshold);
```

We can refine by calling `form.refine()`, so that our refinement loop becomes simply:

```
int numRefinements = 8;
bool printToConsole = true;
for (refNumber = 1; refNumber < numRefinements; refNumber++)
{
  form.refine(printToConsole);
  form.solve();
  exporter.exportSolution(soln, refNumber, numSubdivisions);
}
```

A complete listing for the Stokes cavity driver using `StokesVGPFormulation` can be found in Section 5.4.2.

5.2.1 `StokesVGPFormulation`: Other Notable Features

A handful of other features in the `StokesVGPFormulation` class are worth noting; it supports

- solving for streamfunctions in 2D,
- imposing outflow conditions (via a zero-traction constraint), and
- checkpointing for mesh and solution objects.

We cover each of these briefly in turn.

2D Streamfunctions. In 2D incompressible flow computations, sometimes one wishes to solve for the streamfunction ϕ, which for a velocity field $\boldsymbol{u}$ is given by the solution to the problem

$$\begin{aligned} \Delta\phi &= \nabla \times \boldsymbol{u}, \quad \Omega \\ \frac{\partial \phi}{\partial n} &= \boldsymbol{u} \times \boldsymbol{n}, \quad \partial\Omega \\ \int_\Omega \phi &= 0. \end{aligned}$$

For 2D problems, `StokesVGPFormulation` sets up a Poisson problem corresponding to the above, driven by the velocity $\boldsymbol{u}$ of the Stokes solution. One may access the streamfunction's `Solution` object via `form.streamSolution()`; the variable ϕ is accessible via `form.streamPhi()`. Thus, to output a stream solution for visualization, we may write the following code.

```
SolutionPtr streamSoln = form.streamSolution();
VarPtr phi = form.streamPhi();
streamSoln->solve();
FunctionPtr phiSoln = Function::solution(streamSoln, phi);
HDF5Exporter streamExport(streamSoln->mesh(),
                          "ch05-stokes-stream");
streamExport.exportFunction({phiSoln}, {"phi"}, refNumber,
                            numSubdivisions);
```

A plot of the streamfunction for Stokes cavity flow on a quartic mesh after 8 refinements can be seen in Figure 5.2.

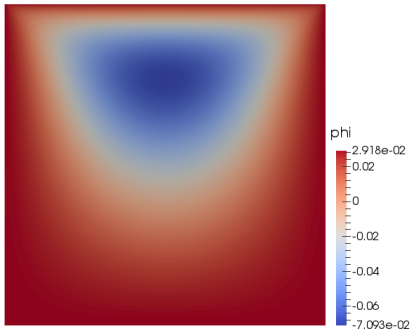


Figure 5.2: Streamfunction for the Stokes cavity flow problem: ϕ solution on quartic mesh after 8 refinements.

Outflow Conditions. In solving incompressible flow problems, there is some art involved in setting appropriate conditions at outflow—and in ensuring that outflow regions are sufficiently far from regions of interest in the domain. One common choice at outflow boundaries is to impose *zero-traction* conditions. As mentioned in passing in Chapter 3, the flux $\widehat{\boldsymbol{t}}_n$ in the ultraweak VGP formulation is sometimes referred to as a pseudo-traction; one can impose zero conditions on this variable at the outflow and achieve reasonable results. However, it may be preferable to impose a zero condition on the physical traction rather than the pseudo-traction. The physical traction is defined as

$$\begin{aligned}\boldsymbol{t}_{\text{phys}} &= \mu\left(\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^T\right)\boldsymbol{n} - p\,\boldsymbol{n} \\ &= \left(\boldsymbol{\sigma} + \boldsymbol{\sigma}^T\right)\boldsymbol{n} - p\,\boldsymbol{n},\end{aligned}$$

where $\boldsymbol{n}$ is the unit normal on the outflow boundary. `StokesVGPFormulation` provides a method, `addOutflowCondition()`, which imposes a zero condition on either the physical traction $\boldsymbol{t}_{\text{phys}}$ or on the pseudo-traction $\widehat{\boldsymbol{t}}_n$; which condition is imposed is controlled by a boolean argument, `usePhysicalTraction`. Thus, once we have a formulation object `form` constructed and have called `form.initializeSolution()`, if we have a `SpatialFilter` object `outflow`, we may do the following to impose a physical traction:

```
bool usePhysicalTraction = true;
form.addOutflowCondition(outflow, usePhysicalTraction);
```

It is worth noting that since the physical traction is not a solution variable in our formulation, this condition is imposed using penalty constraints. By contrast, if we impose an outflow condition using the pseudo-traction, we simply impose Dirichlet conditions on the $\widehat{\boldsymbol{t}}_n$ variables.

Checkpointing. It is common, in scientific computations of many kinds, to write the computational state to disk at various points during a run—this is known as a *checkpoint*. This allows later runs to pick up where a prior run left off. One generally writes checkpoints multiple times during the run, not just at the end of the run, to guard against possible interruptions of the driver. (Examples of interruptions include running out of time or memory in an allocation, as well as hardware and software failures of many kinds.) `StokesVGPFormulation` takes advantage of features in `Solution` and `Mesh` to write checkpoints and restore state from previously written checkpoints.

Once one has initialized the solution in the `form` (whether one has solved and/or refined or not), one may write a checkpoint as follows.

```
int checkpointNumber = 0;
ostringstream checkpointPrefix;
checkpointPrefix << "stokes-checkpoint-";
checkpointPrefix << checkpointNumber;
form.save(checkpointPrefix.str());
```

This example also shows how to create a filename prefix that depends on an integer checkpoint number, so that one may store a series of checkpoints. To load from a checkpoint, one simply does the following:

```
form.load(checkpointPrefix.str());
```

Note that, in the present implementation, the checkpoint only stores solution data and mesh information—it does not store boundary conditions, for example. Calls to `addOutflowCondition()`, for example, should still be made after loading from a checkpoint.

5.3 The `NavierStokesVGPFormulation` Class

The `NavierStokesVGPFormulation` class makes use of the `StokesVGPFormulation` class, and provides many of the same features in the context of the Navier-Stokes VGP formulation derived in Chapter 4. In addition, it supports features useful for performing Newton iterations. Because the Navier-Stokes bilinear form refers to the background flow, and the background flow is defined on a mesh, constructors for `NavierStokesVGPFormulation` require a `MeshTopology` to be provided, and a field polynomial order to be selected. (This is in contrast with the Stokes case, where we were able to construct the formulation, and only later call `initializeSolution()`, providing a `MeshTopology` at that point.) Supposing that we have an appropriately defined `MeshTopology`, `meshTopo`, we may construct a `NavierStokesVGPFormulation` as follows:

```
int spaceDim = 2; // 3D also supported
double Re = 100;
bool useConformingTraces = true;
int polyOrder = 4;
int delta_k = 2;
auto form = NavierStokesVGPFormulation::steadyFormulation(
    spaceDim, Re, useConformingTraces, meshTopo, polyOrder,
```

```
    delta_k);
```

When using `NavierStokesVGPFormulation`, we can set up boundary conditions in precisely the same way as we did with `StokesVGPFormulation`; in contrast to the driver we implemented in Chapter 4, we do not need to explicitly take the difference between the background flow trace velocity and the velocity we seek to impose on the boundary—the `NavierStokesVGPFormulation` class handles this for us. Similarly, instead of setting up a `Function` object to compute the L^2 norm of the solution and that of the increment, now `NavierStokesVGPFormulation` will do this for us, and we have only to call

```
double l2_soln = form.L2NormSolution();
double l2_incr = form.L2NormSolutionIncrement();
```

We may also now combine the steps of solving for the increment and adding to the background flow by calling `form.solveAndAccumulate()`. Thus the double loop for refinement and Newton iterations now reads:

```
for (refNumber = 0; refNumber <= numRefinements; refNumber++)
{
  do
  {
    form.solveAndAccumulate();
    l2_incr = form.L2NormSolutionIncrement();
    if (rank==0)
      cout << "L^2(increment): " << l2_incr << endl;
  }
  while (l2_incr > newtonThreshold);

  exporter.exportSolution(form.solution(), refNumber,
                          numSubdivisions);
  form.refine(printToConsole);
  energyError = refStrategy->getEnergyError(refNumber);
  l2_soln = form.L2NormSolution();
  // update threshold:
  newtonThreshold = 1e-3 * energyError / l2_soln;
  if (rank==0)
  {
    cout << "L^2(soln): " << l2_soln << endl;
    cout << "Newton threshold: " << newtonThreshold << endl;
  }
}
```

Like the `StokesVGPFormulation`, the `NavierStokesVGPFormulation` supports setting inflow and outflow conditions, as well as both kinds of pressure constraints. The complete code for an adaptive cavity flow solver using `NavierStokesVGPFormulation` can be found in Section 5.4.3.

5.4 Drivers

5.4.1 Poisson Driver

The complete driver for the Poisson problem in Section 5.1 is in a file called `ch05-PoissonHomogeneous.cpp`, which can be found in the Camellia distribution in `manual-examples/ch05-formulations`. We also include it below.

```
//
//   2016 UChicago Argonne.  For licensing details, see LICENSE-Camellia in the licenses directory.
//

#include "Camellia.h"

#include "Teuchos_CommandLineProcessor.hpp"

using namespace Camellia;
using namespace std;

int main(int argc, char *argv[])
{
  Teuchos::GlobalMPISession mpiSession(&argc, &argv); // initialize MPI

  int rank = MPIWrapper::CommWorld()->MyPID();

  int spaceDim = 3;
  bool conformingTraces = true;
  PoissonFormulation form(spaceDim, conformingTraces);
  BFPtr bf = form.bf();

  FunctionPtr f = Function::constant(1.0);
  RHSPtr rhs = form.rhs(f);

  BCPtr bc = BC::bc();
  VarPtr u_hat = form.u_hat();
  SpatialFilterPtr everywhere = SpatialFilter::allSpace();
  bc->addDirichlet(u_hat, everywhere, Function::zero());

  vector<int> elementCounts = {1,1,1}; // x,y,z directions
  vector<double> domainDim = {1.0,1.0,1.0};
  int H1Order = 3;
  int delta_k = 3;
  MeshPtr mesh = MeshFactory::rectilinearMesh(bf, domainDim, elementCounts, H1Order, delta_k);

  // refine mesh uniformly 3 times -- result will be 8x8x8 mesh
  int numRefs = 3;
  for (int i=0; i<numRefs; i++)
  {
    mesh->hRefine(mesh->getActiveCellIDsGlobal());
  }

  SolutionPtr soln = Solution::solution(bf,mesh,bc,rhs,bf->graphNorm());
  soln->solve();

  HDF5Exporter exporter(mesh,"ch05-poisson-homogeneous");
  int numSubdivisions = 30; // coarse mesh -> more subdivisions
  int refNumber = 0;
  exporter.exportSolution(soln, refNumber, numSubdivisions);
```

```
  return 0;
}
```

5.4.2 Stokes Cavity Flow Driver

This chapter's Stokes cavity flow driver can be found in the Camellia distribution in `manual-examples/ch05-formulations`, in `ch05-StokesCavity.cpp`. It can also be found below.

```
//
//   2016 UChicago Argonne.  For licensing details, see LICENSE-Camellia in the licenses directory.
//

#include "Camellia.h"

using namespace Camellia;
using namespace std;

class LidVelocity : public SimpleFunction<double>
{
  double _eps; // interpolation width
public:
  LidVelocity(double eps)
  {
    _eps = eps;
  }
  double value(double x, double y)
  {
    if (abs(x) < _eps)
    {
      return x / _eps;         // top left
    }
    else if (abs(1.0-x) < _eps)
    {
      return (1.0-x) / _eps;   // top right
    }
    else
    {
      return 1;                // top middle
    }
  }
};

int main(int argc, char *argv[])
{
  Teuchos::GlobalMPISession mpiSession(&argc, &argv); // initialize MPI

  int spaceDim = 2; // 3D also supported
  double mu = 1.0;
  bool conformingTraces = true;
  StokesVGPFormulation form = StokesVGPFormulation::steadyFormulation(spaceDim, mu, conformingTraces);

  vector<double> dims = {1.0,1.0}; // domain dimensions
  vector<int> meshDims = {2,2};    // 2x2 initial mesh
  vector<double> x0 = {0.0,0.0};   // lower-left corner at origin

  MeshTopologyPtr meshTopo;
  meshTopo = MeshFactory::rectilinearMeshTopology(dims,
                                                  meshDims, x0);
  int polyOrder = 4;
  int delta_k = 2;
  FunctionPtr f = Function::zero(1); // vector zero
  form.initializeSolution(meshTopo, polyOrder, delta_k, f);

  double eps = 1.0 / 64.0;
  FunctionPtr lidVelocity_x = Teuchos::rcp(new LidVelocity(eps));
  FunctionPtr lidVelocity = Function::vectorize(lidVelocity_x, Function::zero());

  SpatialFilterPtr lid  = SpatialFilter::matchingY(1.0);
  SpatialFilterPtr wall = !lid; // not lid --> wall

  form.addInflowCondition(lid, lidVelocity);
  form.addWallCondition(wall);
  form.addPointPressureCondition({0.5,0.5});

  form.solve();
```

```
  int refNumber = 0;
  SolutionPtr soln = form.solution();
  SolutionPtr streamSoln = form.streamSolution();
  VarPtr phi = form.streamPhi();
  FunctionPtr phiSoln = Function::solution(phi, streamSoln);

  HDF5Exporter exporter(soln->mesh(),"ch05-stokes-cavity-flow");
  HDF5Exporter streamExport(streamSoln->mesh(),"ch05-stokes-stream");

  int numSubdivisions = 30; // coarse mesh -> more subdivisions
  exporter.exportSolution(soln, refNumber, numSubdivisions);

  streamSoln->solve();
  streamExport.exportFunction({phiSoln}, {"phi"},
                              refNumber, numSubdivisions);

  double threshold = 0.2; // relative energy error threshold
  auto refStrategy = form.getRefinementStrategy();
  refStrategy->setRelativeErrorThreshold(threshold);

  int numRefinements = 8;
  bool printToConsole = true;
  for (refNumber = 1; refNumber < numRefinements; refNumber++)
  {
    form.refine(printToConsole);
    form.solve();
    exporter.exportSolution(soln, refNumber, numSubdivisions);
    streamSoln->solve();
    streamExport.exportFunction({phiSoln}, {"phi"},
                                refNumber, numSubdivisions);
  }
  // report final energy error:
  double energyError = soln->energyErrorTotal();
  cout << "Final energy error: " << energyError << endl;

  return 0;
}
```

5.4.3 Navier-Stokes Cavity Flow Driver

This chapter's Navier-Stokes cavity flow driver can be found in the Camellia distribution in `manual-examples/ch05-formulations`, in the file `ch05-NavierStokesCavity.cpp`. It can also be found below.

```
//
//   2016 UChicago Argonne.  For licensing details, see LICENSE-Camellia in the licenses directory.
//

#include "Camellia.h"

using namespace Camellia;
using namespace std;

class LidVelocity : public SimpleFunction<double>
{
  double _eps; // interpolation width
public:
  LidVelocity(double eps)
  {
    _eps = eps;
  }
  double value(double x, double y)
  {
    if (abs(x) < _eps)
    {
      return x / _eps;         // top left
    }
    else if (abs(1.0-x) < _eps)
    {
      return (1.0-x) / _eps;   // top right
    }
    else
    {
      return 1;                // top middle
    }
  }
};

int main(int argc, char *argv[])
{
  Teuchos::GlobalMPISession mpiSession(&argc, &argv); // initialize MPI
  int rank = MPIWrapper::CommWorld()->MyPID();

  double Re = 1e2;

  int spaceDim = 2;
  int meshWidth = 2;
  vector<double> dims = {1.0, 1.0};
  vector<int> numElements = {2,2};
  vector<double> x0 = {0,0};

  MeshTopologyPtr
    meshTopo = MeshFactory::rectilinearMeshTopology(dims,numElements,x0);

  int polyOrder = 4, delta_k = 2;

  bool useConformingTraces = false;
  NavierStokesVGPFormulation form =
    NavierStokesVGPFormulation::steadyFormulation(spaceDim, Re, useConformingTraces,
                                meshTopo, polyOrder, delta_k);

  double eps = 1.0 / 64.0;
  FunctionPtr lidVelocity_x = Teuchos::rcp(new LidVelocity(eps));
  FunctionPtr lidVelocity = Function::vectorize(lidVelocity_x, Function::zero());

  SpatialFilterPtr lid  = SpatialFilter::matchingY(1.0);
  SpatialFilterPtr wall = !lid; // not lid --> wall

  form.addInflowCondition(lid, lidVelocity);
  form.addWallCondition(wall);
  form.addPointPressureCondition({0.5,0.5});
```

```
  double l2_incr = 0;

  int refNumber = 0;
  MeshPtr mesh = form.solution()->mesh();
  HDF5Exporter exporter(mesh,"ch05-navier-stokes-cavity");
  int numSubdivisions = 30; // coarse mesh -> more subdivisions

  double newtonThreshold = 1e-3;
  double energyThreshold = 0.2;
  auto refStrategy = form.getRefinementStrategy();
  refStrategy->setRelativeErrorThreshold(energyThreshold);

  int numRefinements = 8;
  bool printToConsole = true;
  double energyError, l2_soln;
  for (refNumber = 0; refNumber <= numRefinements; refNumber++)
  {
    do
    {
      form.solveAndAccumulate();
      l2_incr = form.L2NormSolutionIncrement();
      if (rank==0) cout << "L^2(increment): " << l2_incr << endl;
    }
    while (l2_incr > newtonThreshold);

    exporter.exportSolution(form.solution(), refNumber, numSubdivisions);

    if (refNumber < numRefinements)
    {
      form.refine(printToConsole);
      energyError = refStrategy->getEnergyError(refNumber);
    }
    else
    {
      energyError = refStrategy->computeTotalEnergyError();
    }
    l2_soln = form.L2NormSolution();
    // update threshold:
    newtonThreshold = 1e-3 * energyError / l2_soln;
    if (rank==0) cout << "L^2(soln): " << l2_soln << endl;
    if (rank==0) cout << "Newton threshold: " << newtonThreshold << endl;
  }
  if (rank==0) cout << "Final energy error: " << energyError << endl;

  return 0;
}
```

Chapter 6

Global Linear Solvers: Direct and Iterative Options

6.1 Solver Interface

Camellia defines a solver interface through the `Solver` class; interfaces to several third-party solvers are built into Camellia. These include the following sparse direct solvers:

- SuperLUDist,
- MUMPS, and
- KLU2 (part of the Trilinos Amesos2 package).

(The MUMPS and SuperLUDist solvers are only available if Trilinos was linked against these when it was built.) Additionally, Camellia offers an iterative solver suitable for adaptively refined problems that start from a coarse mesh; this constructs a geometric multigrid preconditioner and performs either GMRES or conjugate gradient iterations to solve.

In previous chapters, when we had a `SolutionPtr` object called `soln`, we invoked `soln->solve()`. This employs whatever solver is returned by `Solver::getDirectSolver()`; the present implementation uses the first available solver from the list above (KLU2 is always available). This method takes an optional boolean argument, `saveFactorization`. If this is true, then the solver will save factorization information for reuse. This is useful for applications where the stiffness matrix may be unchanged over successive solves; the default value is false, which may reduce memory costs.

6.2 Static Condensation

We have found it is often beneficial to use *static condensation* to reduce the size of the global system by locally eliminating interior degrees of freedom on each element. At present, Camellia supports static condensation for discontinuous field variables[1]—therefore, it is most useful in the context of ultraweak DPG formulations. Static condensation may be used either with the direct solvers described above or with the geometric-multigrid-preconditioned iterative solvers described below.

Mathematically, static condensation proceeds as follows. Suppose that our discrete system is of the form $Kx = F$. The system can be reordered to take the form

$$\begin{pmatrix} K_{11} & K_{12} \\ K_{12}^T & K_{22} \end{pmatrix} \begin{pmatrix} u \\ \hat{u} \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$$

where K_{11} is block diagonal, u represents the degrees of freedom corresponding to field variables, and $\hat{u}$ represents those corresponding to the trace variables. Noting that $u = K_{11}^{-1}(F_1 - K_{12}\hat{u})$, we can substitute this into the equation $K_{12}^T u + K_{22}\hat{u} = F_2$ to obtain the Schur complement system for the trace degrees of freedom:

$$\underbrace{(K_{22} - K_{12}^T K_{11}^{-1} K_{12})}_{=\widetilde{K}} \hat{u} = F_2 - K_{12}^T K_{11}^{-1} F_1.$$

Since K_{11} is block-diagonal, its inversion can be carried out element-wise and in parallel; usually, $\widetilde{K}$ is a significantly smaller matrix and the computational cost of the global solve is reduced.

To set a Camellia solution object `soln` to use static condensation, one may simply call:

```
soln->setUseCondensedSolve(true);
```

It is important to invoke this method prior to creating any `GMGSolver` for `soln`.

[1] One exception worth noting: when a discontinuous field variable has a zero-mean constraint or a point-value imposition, as in pressure constraints for incompressible flow, that variable will not be statically condensed.

6.3 Geometric Multigrid Preconditioned Iterative Solves

The Camellia `GMGSolver` class offers very good scalability,[2] though for smaller problems the benefits are outweighed by the cost of construction. We recommend using an iterative solver for most 3D problems, and for 2D problems with a large number of degrees of freedom (more than 10^6 or so). We also recommend iterative solvers for time-stepping solves in which the system matrix remains fixed, because then the cost of constructing the multigrid operators can be amortized over many solves. For full details on our approach to geometric multigrid preconditioning, see Roberts et al. [16].

Suppose that we have a problem posed on a `Mesh` object named `mesh`, with `Solution` object `soln`, and we would like to perform a conjugate gradient solve to a relative tolerance of 10^{-6}, with a maximum of 1000 conjugate gradient iterations. The simplest way to use `GMGSolver` to do this is as follows:

```
double cgTol = 1e-6;
double maxIters = 1000;
auto solver = GMGSolver::cgSolver(soln, cgTol, maxIters);
soln->solve(solver);
```

At a high level, the `cgSolver()` implementation does the following:

1. Produce a stack of meshes, from coarse to fine (details below).
2. Subject to the constraint that there are at least two meshes in the stack, discard any meshes that have fewer than 2000 degrees of freedom. (The idea is that for meshes this small, the direct solver that we will use at the coarsest level will be quite efficient.)
3. Construct a geometric multigrid V-cycle operator with appropriate prolongation and restriction operators.
4. Construct a `GMGSolver` object that is set to use the conjugate gradient method.

We now briefly describe the construction of the mesh stack in step 1. We start with the finest mesh, which is the mesh used by `soln` to define the problem.

[2] On Argonne's Mira supercomputer, we have run on as many as 32,768 MPI ranks, with 64% of the optimal speedup in the one-element-per-rank limit [16].

We first coarsen maximally in p, to produce the second-finest mesh. We then h-coarsen any active elements in this mesh that have a parent element (i.e., active elements that were produced by a refinement), to produce the next mesh. We repeat this until no elements have any parent elements—i.e., until we have arrived at the root-level mesh. An example mesh stack can be found in Figure 6.1.

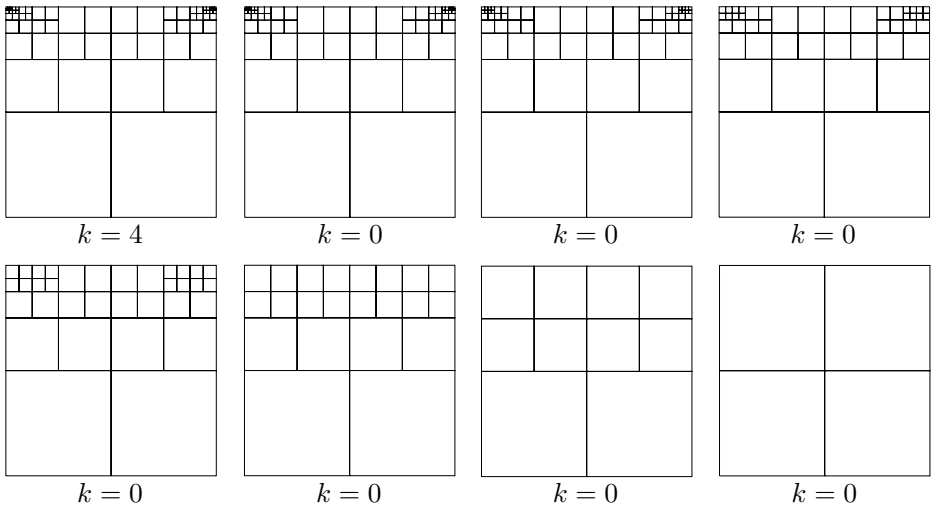


Figure 6.1: Multigrid hierarchy for quartic lid-driven cavity flow with $k = 4$ after six adaptive mesh refinements; at top left is the finest mesh, at bottom right is the coarsest.

6.3.1 Other `GMGSolver` Options

A few other features of the `GMGSolver` class are worth noting. First, there is a static constructor `gmresSolver()`, which works precisely the same way as `cgSolver()`, except that GMRES iterations are performed. In general, CG is suitable for symmetric positive definite (SPD) problems, while GMRES should be used for problems that are not SPD. (Standard DPG stiffness matrices are always SPD—or Hermitian positive definite in the case of complex problems.)

Second, there is a method `meshesForMultigrid()`, which gives direct access to the stack of meshes that `cgSolver()` and `gmresSolver()` employ. One could use this to produce a custom mesh stack, and then construct

a `GMGSolver` using that stack—there is a static constructor, `gmgSolver()`, which takes the mesh stack as an argument. For example, if we have a solution `soln` on a mesh `fineMesh`, would like to use a coarse polynomial order of 1 (instead of the 0 we used above), and would like to discard meshes in the stack that have fewer than 200 elements, we could do the following.

```
int kCoarse = 1;
vector<MeshPtr> meshStack
       = GMGSolver::meshesForMultigrid(fineMesh, kCoarse);
vector<MeshPtr> smallStack;
int numMeshes = meshStack.size();
int meshOrdinal = 0;
for (auto mesh : meshStack)
{
  if (mesh->numActiveElements() > 200)
  {
    smallStack.push_back(mesh);
  }
  else if (numMeshes - meshOrdinal <= 2)
  {
    // then we need to add mesh anyway: need >= 2 meshes
    smallStack.push_back(mesh);
  }
  meshOrdinal++;
}
bool useCG = false; // will use GMRES
GMGOperator::MultigridStrategy multigridStrategy
       = GMGOperator::V_CYCLE; // W_CYCLE also possible
double tol = 1e-6;
int maxIters = 1000;
auto gmgSolver = GMGSolver::gmgSolver(soln, useCG, smallStack,
                                      maxIters,tol,
                                      multigridStrategy);
```

Third, it is possible to control the level of console output that `GMGSolver` produces. Because the output produced by Trilinos's Belos solver imitates the output of the earlier Aztec solver, the option controlling this is `setAztecOutput(int n)`. A zero argument indicates that no output should be produced; an argument of positive n will output the residual every nth iteration, as well as the final residual.

Finally, when operating in CG mode, `GMGSolver` supports computing a condition number estimate of the preconditioned system. The code below demonstrates setting the `gmgSolver` to emit no console output, solving, and

then printing the condition number estimate.

```
gmgSolver->setAztecOutput(0);
gmgSolver->setComputeConditionNumberEstimate(true);
soln->solve(gmgSolver);
// print condest for preconditioned system
double condest = gmgSolver->condest();
cout << "condest: " << condest;
```

Appendix A

Other Features

The present form of this manual is incomplete; it is the first part of what is intended to be a longer manuscript. Due to time constraints, there are several important features of Camellia that we have not covered, or have not covered as fully as we would like. This appendix provides a necessarily brief mention of several of these features, and some notes on how to find out more about them in advance of the publication of a revised version of this manual.

When we refer to source files below, we generally mention the location of the `.cpp` file implementing particular classes. The corresponding `.h` files can be found in the `src/include` directory. Unit tests can be found in the `unit_tests` directory.

Topics that we intend to cover in the main body of the completed manual include the following:

- transient problems,
- implementing custom functions,
- using Camellia for methods other than ultraweak DPG,
- custom refinement strategies,
- unit tests in Camellia,
- visualization,
- common issues with MPI usage in Camellia, and

- rapid implementation of distributed algorithms using Camellia's `MPIWrapper` class.

Topics that we intend to cover in appendices to the completed manual include:

- Camellia's `BasisReconciliation` class,
- Camellia's interface to the MOAB mesh library, and
- exporting matrices for external analysis.

We treat each of these in turn.

A.1 Support For Transient Problems

Camellia has support for both standard time-stepping techniques (such as backward Euler), as well as space-time elements.

Time-Stepping in Camellia

One example demonstrating the use of time-stepping in the context of an ultraweak DPG formulation can be found in `StokesVGPFormulation`, many of whose features we discussed in Chapter 5. The static constructor `timeSteppingFormulation()` will produce a time-stepping Stokes formulation, using a simple graph norm. For more detail on time-stepping DPG schemes, see Führer et al. [10].

Space-Time Elements in Camellia

For any supported element shape, Camellia can create a corresponding space-time extrusion of that element. Tensor-product bases are similarly automatically created.

The space-time support in the present release should be understood as a proof of concept, because of two limitations. First, the temporal extrusions are required to be orthogonal, which means that spatial domains cannot change size or shape in time (an advantage of space-time elements in general is that they provide a natural way to treat such changes in the spatial domain). Second, the implementation does not take special advantage of

the tensor-product structure of the finite element basis functions in the space-time mesh. In general, space-time formulations are more expensive than an equivalent time-stepping formulation, so that it is important to take advantage of whatever optimizations one can for computational efficiency.

For further discussion of space-time elements for DPG, see Truman Ellis's dissertation [9].

A.2 Custom Functions in Camellia

In Chapter 3, we gave an example of a custom `Function` subclass to implement boundary conditions for the cavity flow problem. There, we subclassed `SimpleFunction`, which allowed us to succinctly specify the function's dependence on spatial coordinates. However, it is also possible to create custom functions that depend on the mesh. Similarly, Camellia has rich support for taking derivatives of `Function` objects—one can take advantage of this support by overriding `dx()`, `dy()`, `dz()`, and/or (for space-time support) `dt()`. Standard calculus rules—the product rule and the quotient rule, specifically—are supported by the classes that implement sums, products, and quotients of functions. These facilities are particularly useful in the context of *manufactured solutions*: starting from a set of functions that one wishes to have as the exact solution to a problem, one may use the original PDE to generate the right-hand side and boundary conditions that will yield the desired solution.

For examples of `Function` subclasses that depend on the mesh, see the `hFunction` class (giving the diameter of the current element) and the `MeshPolyOrderFunction` class (giving the H^1 polynomial order of the current element), whose implementations can be found in `hFunction.cpp` and `MeshPolyOrderFunction.cpp`, respectively, under `src/Function`.

For examples of `Function` subclasses that provide support for differential operations, see any of the trigonometric functions implemented in `TrigFunctions.cpp`—also in the `src/Function` directory.

A.3 Other Finite Element Methods

Camellia has rich support for many finite element methods; the intent is to provide support for arbitrary methods. For example implementations

of primal DPG, SUPG, and FOSLS (as well as ultraweak DPG) for the convection-diffusion-reaction problem, see the `ConvectionDiffusionReactionFormulation` class, whose implementation can be found in a `.cpp` file of the same name in the `src/Formulations` directory.

Camellia also has support for the jump terms employed in DG methods—the `BF` class has a method called `addJumpTerm()`, and there is a `Function` subclass called `UpwindIndicatorFunction` that allows specification of the u^- and u^+ terms that arise in DG jump terms. There is a somewhat unpolished example driver that demonstrates the usage of this in the context of a pure convection problem; this can be found in the directory `src/examples/DGAdvectionExample`.

A.4 Custom Refinement Strategies

The `RefinementStrategy` class discussed in Chapter 3 provides mechanisms for refinement strategies driven by an error indicator function of one sort or another. There is built-in support for using the DPG energy error, as well as gradient-based and hessian-based error indicators. One may implement a custom error indicator by subclassing the `ErrorIndicator` class and overriding the `measureError()` method, whose responsibility it is to fill the superclass's `_localErrorMeasures` container. The `GradientErrorIndicator` and `HessianErrorIndicator` classes, whose implemantations can be found in `src/Solution`, are examples of this approach. One can then use the `RefinementStrategy` constructor that takes an `ErrorIndicator` as argument to produce an appropriate refinement strategy.

A.5 Unit Tests

Camellia includes more than 500 unit tests in its `runTests` driver. This is built atop test facilities in the Teuchos package of Trilinos. A few features of the tests driver itself are worth mentioning. First, like essentially all Camellia drivers, it supports being run in parallel via MPI—and for a full test, we recommend running both in serial and on several different counts of MPI ranks. Thus, in the course of development, we often run `runTests` in a sequence something like the following:

```
./unit_tests/runTests
mpirun -np 4 ./unit_tests/runTests
mpirun -np 16 ./unit_tests/runTests
```

We expect all tests to pass in all three cases. Each test identifies a group to which it belongs, as well as the name of the test itself. One can run subsets of tests by specifying group or test names at the command line; one can also match similarly-named groups or tests via glob matches. For example,

```
./unit_tests/runTests --group="*VGPFormulation" --test="*2D*"
```

will run all tests in the groups `NavierStokesVGPFormulation` and `StokesVGPFormulation` that have the string "`2D`" in the test names.

There are a host of other options supported by `runTests`; by running

```
./unit_tests/runTests --help
```

one may see a listing of supported options.

The tests themselves can be found in the `unit_tests` directory. To add a new file implementing further tests, one can duplicate the file `TestsTemplate.cpp` and uncomment and edit the `TEUCHOS_UNIT_TEST()` preprocessor calls to include the desired group name and test name. It is not immediately obvious from the interface how best to write a set of tests that share most of the code, differing only in some parameters. The key to understanding how to do this is to recognize that `TEUCHOS_UNIT_TEST` defines two "hidden" variables, a boolean `success` that defines whether the test passes or fails, and a `Teuchos::FancyOStream` object called `out`. The following code, derived from code in the `StokesVGPFormulationTests.cpp` file, defines one method for testing consistency of the formulation, and calls it from two separate tests: one for 2D, the other for 3D.

```
void testStokesConsistencySteady(int spaceDim, Teuchos::FancyOStream &out, bool &success)
{
  // ...
  double tol = 1e-10;
  TEST_FLOATING_EQUALITY(expectedValue, actualValue, tol);
  // TEST_FLOATING_EQUALITY sets success to false if: abs(expectedValue - actualValue) > tol
  // here's explicit code for that:
  if (abs(expectedValue - actualValue) > tol)
  {
    success = false;
  }
  out << "Here's some output I want to see if the test fails.\n"
}
TEUCHOS_UNIT_TEST( StokesVGPFormulation, Consistency_2D_Steady )
{
  int spaceDim = 2;
  testStokesConsistencySteady(spaceDim,out,success);
}

TEUCHOS_UNIT_TEST( StokesVGPFormulation, Consistency_3D_Steady_Slow )
{
  int spaceDim = 3;
  testStokesConsistencySteady(spaceDim,out,success);
}
```

A.6 Visualization

In several examples in Chapters 3, 4, and 4, we used the `HDF5Exporter` class to export solutions and functions to a format that ParaView can read. This is the principal strategy we employ for visualization in Camellia; we may then do any post-processing required in ParaView itself.

For 2D visualizations, however, Camellia also includes the class `GnuPlotUtil`. This includes a set of utilities that will output text files suitable for processing with gnuplot. This can be a convenient way to visualize a 2D mesh, for example, particularly since there is a facility for including numeric labels corresponding to cell IDs. Supposing that we have a 2D `Mesh` object called `mesh`, the following code will output two files, `mesh` and `mesh.p`, to the working directory.

```
string exportName = "mesh";
int numPointsPerEdge = 2;
bool labelCells = true;
string meshColor = "black";
MeshTopologyPtr meshTopo = mesh->getTopology();
GnuPlotUtil::writeExactMeshSkeleton(exportName,meshTopo.get(),
                                    numPointsPerEdge,
                                    labelCells, meshColor);
```

One can then run

```
gnuplot mesh.p
```

to generate an EPS representation of the mesh in `mesh.eps`. A sample mesh plot—of the Stokes cavity flow problem after 3 refinements—can be found in Figure A.1.

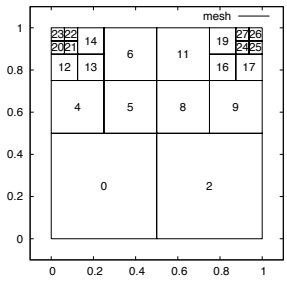


Figure A.1: A figure generated by gnuplot using Camellia's `GnuPlotUtil` facility: cavity flow mesh after 3 adaptive refinements, with cells labelled.

A.7 Common Issues with MPI

Most Camellia drivers—including the examples we have provided here—can be run on several MPI ranks without special consideration or modification. It is important to be aware of a few facts, however, regarding Camellia's parallel operation, to avoid crashes or hangs when running in parallel:

- By default, Camellia's `MeshTopology` is distributed during the initialization of a `Mesh` object based on it. After that point, not all cells will be available on all MPI ranks. Locally-owned cells can be accessed through `Mesh`'s `cellIDsInPartition()` method; it is guaranteed that both the geometry of local cells and that of their immediate neighbors will be available. If an attempt is made to access an unavailable cell, `MeshTopology` will throw an exception.

- Similarly, solution values are distributed according to the same partitioning as the mesh. Attempts to access off-rank solution data may result in zero values or run-time exceptions, depending on the context. If a driver has inconsistent values for some quantity of interest, it is worth checking that only local solution values are being accessed.

- Several Camellia methods perform MPI communication. For these methods, it is important that they be run on every MPI rank within the

relevant MPI communicator (by default, this is `MPI_COMM_WORLD`). One common mistake is to enclose console output within an `if (rank == 0)` guard, and within that `if` block to call some MPI-communicating method. If a driver hangs, this is a pattern to look out for.

A.8 Distributed Algorithms using Camellia's `MPIWrapper`

MPI provides several useful kernels for distributed computation. However, the interface is such that one often ends up writing fairly similar code repeatedly when implementing algorithms that use MPI, and since there is often some reliance on pointer arithmetic, the development process is somewhat error-prone. Camellia has a class, `MPIWrapper`, which aims to provide mechanisms for efficient development of MPI-based algorithms, particularly those that involve C++ Standard Template Library (STL) containers such as `vector` and `map`.

To take one example, the `HessianErrorIndicator` will use the Hessian of the finite element solution polynomial if it is quadratic or higher order; otherwise, it will use a finite difference approximation based on gradients in neighboring elements. As mentioned in the previous section, Camellia distributes solution data, so that neighboring elements' solution data might not be local. Therefore, the implementation must request gradient values for any non-local cells.

The following code creates a `set` that includes all remote neighbor IDs for a `Mesh` object named `_mesh`:

```
auto myCells = &_mesh->cellIDsInPartition();

set<GlobalIndexType> allNeighbors;
for (GlobalIndexType myCellID : *myCells)
{
  CellPtr cell = meshTopo->getCell(myCellID);
  auto neighbors = cell->getActiveNeighborIndices(meshTopo);
  allNeighbors.insert(neighbors.begin(),neighbors.end());
}
set<GlobalIndexType> remoteNeighbors;
for (GlobalIndexType cellID : allNeighbors)
{
  if (myCells->find(cellID) == myCells->end())
  {
```

```
    remoteNeighbors.insert(cellID);
  }
}
```

Now that we have the remote neighbors, we want to request their gradient values. We want to send a request to the owning MPI rank; the request should contain both the cell ID whose gradient value we want, and the requesting MPI rank (so that the owning MPI rank knows where to send the gradient). Cell IDs in Camellia have type `GlobalIndexType` (an integer type, at present defined as `int`), while MPI ranks are `int`-valued. We would like to create a table that has an entry for each remote rank; each entry consists of a list of requests, which are (`int`,`GlobalIndexType`) pairs. The STL `map` type allows us to create lookup tables; the `vector` type allows us to create variable-length arrays; the `pair` type allows us to create ordered pairs. Thus an appropriate container type for our requests is

```
map<int,vector<pair<int,GlobalIndexType>>>;
```

the code for constructing and populating this container follows.

```
Epetra_CommPtr Comm = _mesh->Comm();

int myRank = Comm->MyPID();
map<int,vector<pair<int,GlobalIndexType>>> requests;
for (GlobalIndexType remoteCellID : remoteNeighbors)
{
  int rank = _mesh->partitionForCellID(remoteCellID);
  requests[rank].push_back({myRank,remoteCellID});
}
```

Now, before we send and receive, we need to establish an appropriate container for the requests we receive. This will be a list of (requestor, cell ID) pairs—thus an appropriate type for the received requests will be exactly the type stored in each entry of the requests table: `vector<pair<int,GlobalIndexType>>`. The code below sends and receives, placing the received requests in the `requestsReceived` container.

```
vector<pair<int,GlobalIndexType>> requestsReceived;
MPIWrapper::sendDataVectors(Comm, requests, requestsReceived);
```

Now that each MPI rank has received all requests for its rank-local values, it can populate a container with the values and send to the recipient. What we want to send for each request is now a pair of (cellID, gradient); the gradient

is vector of real values. In this case, the natural thing might seem to be to send a `vector` of ordered pairs of type (`GlobalIndexType`, `vector<double>`). The problem with this is that `MPIWrapper::sendDataVectors` requires the data type of vector entries to have fixed length, and `vector<double>` violates this requirement. What we do instead, therefore, is send an ordered sequence of entries, one for each component of the gradient. The type of each entry is then (`GlobalIndexType`, `double`), and the appropriate container type for the responses is

`map<int,vector<pair<int,double>>>`;

the code below populates this container, supposing that there is already a lookup table called `gradients` that contains the gradients of rank-local cells (stored as `vector<double>` entries in a `map` whose keys are the cell IDs), and performs the send and receive:

```
map<int,vector<pair<GlobalIndexType,double>>> responsesToSend;
vector<pair<GlobalIndexType,double>> responsesReceived;
for (pair<int,GlobalIndexType> request : requestsReceived)
{
  int remotePID = request.first;
  GlobalIndexType myCellID = request.second;
  vector<double> gradient = gradients[myCellID];
  for (double gradient_comp : gradient)
  {
    responsesToSend[remotePID].push_back({myCellID,
                                          gradient_comp});
  }
}
MPIWrapper::sendDataVectors(Comm, responsesToSend,
                            responsesReceived);
```

To add the remote gradients to the `gradients` lookup table, we may do the following:

```
for(pair<GlobalIndexType,double> response : responsesReceived)
{
  GlobalIndexType remoteCellID = response.first;
  double gradient_comp = response.second;
  gradients[remoteCellID].push_back(gradient_comp);
}
```

A.9 The `BasisReconciliation` Class

A key component in any finite element code is the management of degree-of-freedom *connectivities*: how local discretizations of a variable on each element relate to each other on inter-element interfaces. Often, approaches to this problem rely on particular features of a basis implementation that are known *a priori*. This makes it difficult to change basis implementations, and particularly difficult to support multiple types of bases. Moreover, implementing the rules that govern reconciliation of incompatible elements (as when polynomial orders differ on an interface, or there are hanging nodes) can be tedious and error-prone.

Camellia takes a different approach in its `BasisReconciliation` class. The class is responsible for computing at runtime the relationship between discretizations on neighboring elements; it can cache the results so that they do not need to be computed more than once, making the additional cost of the approach negligible. The principal requirement when using `BasisReconciliation` is that one of the bases be strictly finer than the other—if so, the class can determine an exact representation of the coarse basis in terms of the fine basis on the shared interface. Arbitrary levels of h refinement are supported. Camellia uses `BasisReconciliation` both in its implementation of the minimum rule (in the `GDAMinimumRule` class) and in its construction of the prolongation operator for geometric multigrid (in the `GMGOperator` class).

A.10 Importing Meshes Using MOAB

The examples in this manual have all used meshes with simple geometry constructed dynamically by Camellia. In practice, mesh geometries are often more complex, and are defined using meshing packages such as CUBIT [2]. The MOAB library provides mechanisms for working with many standard mesh formats, including the ExodusII format used by CUBIT. When one builds Camellia with MOAB, then one can import meshes in essentially any format supported by MOAB. There is a simple example driver, `MOABReader`, demonstrating this capability; this is available in the `examples` directory of the Camellia distribution.

A.11 Exporting Matrices for External Analysis

Often when studying finite element methods, it is useful to study the system matrices using a tool such as Octave or MATLAB. Camellia provides a mechanism for exporting the system matrix as well as the right-hand side in a MATLAB-compatible format. If one has a `Solution` object `soln`, before `solve()` is called, one may write

```
soln->setWriteMatrixToFile(true, "A.dat");
soln->setWriteRHSToMatrixMarketFile(true, "b.dat");
```

Then, during `solve()`, `Solution` will write the system matrix to `A.dat` in a sparse matrix format; it will write the right-hand side to `b.dat`. In Octave or MATLAB, one may then write:

```
octave:1> load A.dat; A = spconvert(A);
octave:2> load b.dat;
```

One may then perform any operations of interest on `A` and `b`.

Bibliography

[1] Roscoe A. Bartlett. Teuchos::RCP beginner's guide. Technical Report SAND2004-3268, Sandia National Laboratories, http://trilinos.sandia.gov/RefCountPtrBeginnersGuideSAND.pdf, 2010.

[2] Ted Blacker, Steven J. Owen, Matthew L. Staten, Roshan W. Quadros, Byron Hanks, Brett Clark, Ray J. Meyers, Corey Ernst, Karl Merkley, Randy Morris, Corey McBride, Clinton Stimpson, Michael Plooster, and Sam Showman. CUBIT: Geometry and mesh generation toolkit 15.2 user documentation. Technical Report SAND2016-1649 R, Sandia National Laboratories, 2016.

[3] R. Cai, Z.and Lazarov, T.A. Manteuffel, and S.F. McCormick. First-order system least squares for second-order partial differential equations. I. *SIAM J. Numer. Anal.*, 31:1785–1799, 1994.

[4] C. Carstensen, L. Demkowicz, and J. Gopalakrishnan. Breaking spaces and forms for the DPG method and applications including Maxwell equations. *Computers & Mathematics with Applications*, 72(3):494 – 522, 2016.

[5] Bernardo Cockburn and Jayadeep Gopalakrishnan. The derivation of hybridizable discontinuous Galerkin methods for Stokes flow. *SIAM Journal on Numerical Analysis*, 47(2):1092–1125, 2009.

[6] L. Demkowicz and J. Gopalakrishnan. A class of discontinuous Petrov-Galerkin methods. Part I: The transport equation. *Comput. Methods Appl. Mech. Engrg.*, 199:1558–1572, 2010. See also ICES Report 2009-12.

[7] L. Demkowicz and J. Gopalakrishnan. A class of discontinuous Petrov-Galerkin methods. Part II: Optimal test functions. *Numer. Meth. Part. D. E.*, 27(1):70–105, January 2011.

[8] L. Demkowicz and J. Gopalakrishnan. A primal DPG method without a first-order reformulation. *Computers & Mathematics with Applications*, 66(6):1058 – 1064, 2013.

[9] Truman Everett Ellis. *Space-Time Discontinuous Petrov-Galerkin Finite Elements for Transient Fluid Mechanics*. PhD thesis, University of Texas at Austin, 2016; available at https://github.com/trumanellis/Papers/blob/master/Dissertation/Dissertation.pdf.

[10] Thomas Führer, Norbert Heuer, and Jhuma Sen Gupta. A time-stepping DPG scheme for the heat equation, 2016; available at http://arxiv.org/abs/1607.00301.

[11] J. Gopalakrishnan and W. Qiu. An analysis of the practical DPG method. Technical report, IMA, 2011. submitted.

[12] H.K. Moffatt. Viscous and resistive eddies near a sharp corner. *Journal of Fluid Mechanics*, 18(1):1–18, 1964.

[13] Nathan V. Roberts. *A Discontinuous Petrov-Galerkin Methodology for Incompressible Flow Problems*. PhD thesis, University of Texas at Austin, 2013.

[14] Nathan V. Roberts. Camellia: A software framework for discontinuous Petrov-Galerkin methods. *Computers & Mathematics with Applications*, 2014.

[15] Nathan V. Roberts, Tan Bui-Thanh, and Leszek F. Demkowicz. The DPG method for the Stokes problem. *Computers and Mathematics with Applications*, 2014.

[16] Nathan V. Roberts and Jesse Chan. A geometric multigrid preconditioning strategy for DPG system matrices. *Computers & Mathematics with Applications*, 2016 (submitted); available at http://arxiv.org/abs/1608.02567.

[17] Nathan V. Roberts, Leszek Demkowicz, and Robert Moser. A discontinuous Petrov–Galerkin methodology for adaptive solutions to the incompressible Navier–Stokes equations. *Journal of Computational Physics*, 301:456 – 483, 2015.

Argonne Leadership Computing Facility
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC